OMSI Pascal NEWSLETTER

NUMBER 1 | OREGON SOFTWARE | NOVEMBER 1980

Our software support and how it works

You've just received your OMSI Pascal software, and you can't get the compiler to start up; or perhaps you've run into a mysterious problem with your program three days before a critical project deadline. What help can you expect from Oregon Software?

We know the trauma that can be generated by these, or similar, circumstances. Computer programmers ourselves, we've designed our support policy to help the people in the trenches in every way that we reasonably can.

Formally, our software support is a fixed-term contract with you. The license fee for your Pascal system includes the first year of support. Near the end of that year, we'll notify you that your support is about to expire. We'll offer to renew the contract, one year at a time. Our current price to extend Pascal-1 support for a year is $600 (USA).

Our support policy is explained in detail below, but here's an outline of the major benefits of support:

1) Telephone assistance. Call us for a quick cure.
2) Formal, written response to all problems, suggestions, and comments received in writing.
3) A no-cost update, upon your written request, to the latest revision of your Pascal product.
4) This newsletter, which will contain status reports on all of our Pascal programs, announcement of new versions, and various technical articles.

Telephone access

If you are having difficulties installing your Pascal system, if a paragraph in the manual doesn't make sense, if you think you've uncovered a bug and want to know whether we've found it already — telephone us. We'll put you in touch with our programmers (often the person who wrote the part that's giving you trouble). We'll answer your question then and there if we can. Often, our response is something like, "Yes, that is a bug, it's fixed in V1.2F. We'll send you a copy. Do you want it shipped air freight?"

When you call, it will save time if you know your site number. To find your site number, look at the headline of any Pascal program listing. The site number is on the right side. Finally, if the result of the phone call is that you request (in writing) an update, your company may require a purchase order, even though the update is at no cost.

If your problem is too difficult to handle via the phone, we will always ask for a written description, with examples, a terminal printout, copies of the program and command files, etc.

Written trouble reports

For more complex problems, or for those

Continued on Page 2

In this issue...

Oregon Software support policy — Page 1

A log of changes in Pascal-1 V1.2 — Page 3

Pascal calling FORTRAN — Page 5

Technical Troubleshooting

- When the compiler crashes — Page 6
- Three RSX solutions — Page 7
- RSTS/E V7, RT-11 V4 — Page 9

Call for articles — Page 10

Support

that are not holding up your major development, we ask for written descriptions of the trouble. Written reports allow us to reproduce the problem on our computers to ensure correct diagnosis and repair.

In a written problem description, please be sure to include your site number and the version numbers of your Pascal system and of your operating system. Send us the simplest program that demonstrates your problem. If you must send a program larger than one page, send it in machine-readable form and include all required data and control files.

Our goal is to respond to each written trouble report within 30 days of receipt. We haven't always succeeded, so we're revising and streamlining our procedures. We're also preparing a new Trouble Report form, which will be sent to you soon.

Updates and revisions

We are continually revising our Pascal systems to correct bugs, to enhance capabilities, to adapt to new operating systems from Digital Equipment Corp. We incorporate the improvements into new versions of the product.

Each version has a unique name that consists of the software product number, the major release number, and the minor revision letter. For example, the current version of OMSI Pascal-1 is V1.2F. The designation "V1.2F" means that the software product, OMSI Pascal-1, has undergone one substantial product change (from V1.1 to V1.2 in December 1979) and a total of six minor revisions to date.

Every time we revise a product, you are entitled to a copy of the revised version. We call this an update. Successive issues of this newsletter will describe changes in the product and list the version numbers of our latest products.

To compare the current version number with the version number of your system, check the headline of any program listing. To receive a copy of an update, simply send us your request in writing.

You will not be charged for updates on magtape or floppy disks. You will be charged list price for updates on other media, such as RK05 and RL01. Or, you may enclose the medium with your request for an update, and we will return it with your software. (Remember, you may need a purchase order.)

Oregon Software will pre-pay shipment via UPS. At your request, we will ship air freight collect. All international updates must be shipped by air freight, collect.

The Newsletter

As noted above, the newsletter will describe changes in our products. The newsletter also will contain technical articles written by our staff and by users. We also will announce policy statements through the newsletter. Our goal is to have this publication answer many of your questions before you ask them.

Out of Support?

Occasionally, users will call for help, only to discover that their support has lapsed. The fee for a return to support is $750 (USA).

The Log: Pascal-1 V1.2

Since the release of Pascal-1 V1.2 in December 1979, Oregon Software has continued to fix bugs and add new features. When enough small changes accumulate, or when a significant bug is fixed, we release an update of V1.2 incorporating the improvements. Each updated release is identified by a revision letter. We are currently releasing V1.2F.

This log describes the significant changes we have made to V1.2 — the most important corrections and the most useful new features — and so is not complete. A copy of our complete internal log is available on request. Each issue of the newsletter will continue to outline our revisions.

To use this log, you should first determine what release of V1.2 you have. (That release number is printed on the headline of all program listings.) Then you should review the log to determine the changes made since the release of your version. If you desire an update, request one in writing. (See our support policy, described on the front page of this newsletter.)

This log is divided into four sections. The first describes changes made to all operating systems. The others describe the changes that are specific to each operating system. The version listed is the one in which the change was first made. Succeeding versions, of course, include the change also.

Changes to all systems

V1.2B corrected this problem:

**Compiler generates ADDB instructions

For certain array references indexed by a user-defined scalar type, the compiler generated an ADDB instruction. This instruction is not defined and caused errors at assembly time.

V1.2C corrected this problem:

**Incorrect code for standard functions

In complex cases involving the Abs( ) or Sqr( ) functions and comparison operators, incorrect code was generated, destroying the value of an operand.

V1.2D corrected these problems:

**Debugger will not assign set of char

The Debugger would not accept commands to assign values to variables of type 'set of char', giving an error message.

**Compiler generates incorrect code for Sqr( )

The FIS/EIS/SIM version of the compiler generated incorrect code to push Real arguments for the Sqr( ) function.

**Revision of Write(Real) routine
The floating-point output routine was revised to print seven significant digits and to correct a rounding problem that caused a -0 value to be written.

V1.2F corrected these problems:

**Compiler generates incorrect code for Round( )

In certain cases involving a user-defined function as the argument, code to assign the value of the Round( ) standard function was generated incorrectly.

**Compiler crashes on array syntax error

The compiler crashed when processing an erroneous array declaration using parentheses rather than brackets.

Changes to RSTS/E

V1.2B corrected these problems:

**Debugger gives Link bad module error

One Debugger module was distributed

Continued on Page 4

RSTS/E

containing an illegal object record, making the Debugger unusable.

**Magtape long record blocking problem

Records longer than 512 bytes written to magtape were incorrectly extended to be a multiple of 512 bytes.

V1.2F corrected this problem:

**Change in V7 Link causes improper allocation

The Link program distributed with RSTS/E V7 redefined the meaning of the program high limit word (a difference of two bytes), causing the last word of the program to be overwritten in certain cases.

Changes to RT-11

V1.2D corrected these problems:

**Problem with direct access files

In certain cases, files opened with Rewrite( ) would not perform Seek( ) operations correctly after Put( ) or Write( ) operations.

**File open problems after .CHAIN

A .CHAIN operation leaves channel 15. open, causing the next file Reset( ) or Rewrite( ) to fail.

**FPP and FIS hardware exceptions not handled

FPP and FIS exception traps were not caught by the Pascal library routines, causing an RT-11 monitor error instead.

**Fast assembler (MAC.SAV) does not operate

Several problems caused the MAC assembler to fail in all cases.

**Last character of Text file contains

The last (odd) byte of odd-length output transfers was not set to zero and wrote an uninitialized character.

Changes to RSX

V1.2B corrected this problem:

**Characters transferred as 8-bit data

The library was revised to transfer all 8 bits of character data, rather than stripping the parity bit.

V1.2C corrected this problem:

**Problems writing empty or blank lines

Writing several consecutive blank lines caused a status error, and the error was not reported correctly because the error-reporting routine attempted to write another blank line.

V1.2D corrected these problems:

**Problems with the Close( ) procedure

The closing of large numbers of files caused available memory to be exceeded and in some cases would cause problems with other files.

**Available memory exceeded on task initialization

Programs with checkpointing disabled and no heap extension caused immediate failure on initiation. A special error message was added to identify this problem.

**Exit with status added to compiler and tasks

The compiler was modified to exit with an error status on compilation errors to allow command files to detect such errors. The facility is also available to Pascal programs as the external procedure Exitst( ).

Continued on Page 5

RSX

V1.2F corrected these problems:

**Problems with resident Pascal libraries

Problems were corrected with the installation of the Pascal library as a resident library and to permit the installation of compiled Pascal modules in resident libraries.

**Ctrl/Z returns extra Eoln( )

The Ctrl/Z signal caused a superfluous Eoln( ) indication.

**Compiler counts form feed as extra line

The compiler counted FF (form feed) characters as an additional line, causing line numbers in the listing to disagree with the system editors.

FORTRAN calling Pascal

It is possible, if tricky, to call global Pascal procedures and functions from FORTRAN main programs. The following instructions and routines should help:

To call a procedure with no parameters, no interface routine is necessary, just CALL.

A procedure with parameters should use the following routine as embedded code immediately prefacing the Pascal procedure:

```
(*$C
$ver==     0        ;define globals
$begin==   0        ;to avoid errors

forpas:: ;FORTRAN entry name

      mov  (r5)+,r0          ;arg count
      bic  #255*256,r0       ;(low byte)
1$:   mov  (r5)+,-(sp)       ;stack args
      dec  r0
      bne  1$                ;loop
*)
(* fall into Pascal procedure *)

procedure . . .
```

Calling a Pascal function from FORTRAN is a bit more complex:

```
(*$C
$ver==     0        ;define globals
$begin==   0        ;to avoid errors

forfun::   ;FORTRAN entry name

      clr  -(sp)             ;open space
      clr  -(sp)             ;for return

      mov  (r5)+,r0          ;five lines
      bic  #255*256,r0       ;omitted for
1$:   mov  (r5)+,-(sp)       ;functions
      dec  r0                ;without
      bne  1$                ;parameters

      jsr  pc,pasfun         ;call Pascal

      mov  (sp)+,r0          ;copy return

      mov  (sp)+,r1          ;value

      rts  pc
*)
```

Some general rules:

1) Use the /E switch when you compile the Pascal module;

2) Include the Pascal library in the Link or Task Build step (for RSTS/E, use the special RTSLIB library file);

3) Remember that all parameters must be VAR parameters.

And a general caveat: The Pascal routines should not attempt to perform any I/O, or to allocate any dynamic memory via the New( ) procedure. These operations would reference values within the support library that have not been initialized.

Technical Troubleshooting

When Pascal crashes

An uncommon but frustrating problem is a compiler crash: a monitor error message, or a machine trap during compilation of a Pascal program. We agree that a crash is not an appropriate indicator of an error. However . . .

If you find yourself in this situation, remember one thing. In our experience, the compiler has never crashed while compiling a valid Pascal program. This means that you should be able to examine your program, correct a spelling or syntax error, and cure the problem immediately.

We have two suggestions to help you locate the source of a compiler crash.

First, correct any program errors that are identified by the compiler. Sometimes the "ripple effects" of early errors cause subsequent errors to have disastrous consequences.

The second suggestion, if the problem persists, is for you to print the listing file produced by the compiler up to the point of the crash. This will help identify the location of the program error, because the part of the program that is printed is the part that does not cause the crash. Carefully examine the source program immediately following the point where the listing ended; the error should be in the next few lines. (The listing file is buffered by the compiler, so that this method will not precisely locate the error.)

If you find a program that causes the compiler to crash, we will appreciate hearing about the details so that we can correct future versions of the compiler to give an appropriate error message instead.

Pascal runs under TSX-Plus

TSX-Plus, a multi-user system that emulates RT-11, now operates with Pascal-1 after a problem in an early release that caused the compiler to crash during any real arithmetic procedure.

A spokesman for S & H Computer Systems, which produces TSX-Plus, said the problem occurred in a pre-release version that was distributed to a few selected customers. The problem involved the floating-point simulator.

All recipients of the early release should have been automatically updated with V1.4 of TSX-Plus, which corrects the problem, the spokesman said. Any customers who did not receive V1.4 should contact S & H Computer Systems in Nashville, Tenn., phone 615-327-3670.

S & H's TSX system has had no difficulty running Pascal.

RSX start failure

If you run a program on RSX using V1.2C or previous, the program may abort with a memory protect violation if the task is not checkpointable.

If your program is called TEST, for instance, try running the program with the command "RUN TEST/CKP=YES". This should enable checkpointing (swapping), and the program should run correctly. Be sure the checkpoint file is active (use the ACS command). If the program does not run correctly at this point, your problem lies elsewhere.

Pascal-1 tasks must be checkpointable, because they use the "extend task" monitor call to request additional memory for the task as it is needed. If the task is not checkpointable, this request for more memory will fail. When this happens, Pascal will print the error message: "Not enough available memory". When a Pascal task first starts up on RSX systems, it allocates 2K words of memory for the stack. If this allocation fails, the program attempts to write the error message described above.

However, this particular error happens before Pascal has finished initializing the file system, and Pascal gets an error trying to print the original error message. This leads to a recursive error condition that eventually causes the memory protect violation.

We strongly recommend that you set the Task Builder default to be checkpointable. On busy systems it is much more efficient to have tasks swapped out of memory if they do not need to run. Appendix E in the "Task Builder Reference Manual" describes how to change Task Builder defaults. At the same time, you may want to change the default setting for the floating point (/FP) option, since this option is required by all Pascal tasks running on machines with floating-point processors.

If you do not want a Pascal task to be checkpointable, or if the "extend task" option was not selected when your system was built, you must allocate sufficient dynamic memory for your Pascal task by extending the size of the program section (PSECT) called $$HEAP. Normally, the size of this PSECT is zero, and this tells Pascal to try to extend the size of the task. If you extend the size of $$HEAP with the EXTSCT option when you build your program, Pascal will use the memory in $$HEAP for the stack and all of the data associated with the program. All fixed and local data will be allocated there, as well as file buffers and other dynamic memory allocated via the NEW procedure.

You should allow plenty of room in $$HEAP, and if you want to use the maximum amount of memory possible, extend $$HEAP until the size of your Pascal task reaches the 32K word limit. You should try extending the heap to the maximum size whenever you get the "Not enough memory" error message.

Another approach is to specify the size of $$HEAP from within your Pascal program by including the following code at the beginning of your program:

```
(*$C
.PSECT $$HEAP,D,OVR ;Dynamic memory
.BLKW  4000.        ;Reserve 4000 words
.PSECT              ;Return to blank PSECT
*)
```

This code will automatically allocate about 4K words of memory for the heap, and you will not need to use the EXTSCT option when the task is built.

This problem has been fixed in later releases, and an appropriate error message is printed.

RSX files run out of memory

If you have an RSX Pascal program that opens and closes lots of files, it may, after running a long time, run out of memory. In some cases, the close routine

Continued on Page 8

for Pascal-1 V1.2C or previous for RSX does not correctly deallocate the record buffer associated with the file.

Since the record buffer is usually only a few dozen words long, this problem only causes trouble in programs that open and close hundreds of files. For such programs you can use a Task Builder option to patch the Pascal file close routine to prevent the error. If your program is called TEST, for instance, the program should be built with the following commands:

```
>TKB
TKB>TEST=TEST,LB:1,1PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>GBLPAT=TEST:$CLOS+50:1423
TKB>//
```

The GBLPAT option patches location $CLOS+50 in the close module to contain the value 1423. This patch is needed only for programs using large numbers of files. This problem has been fixed in V1.2D and later versions of RSX.

RSX resident libraries

"On RSX and IAS systems when I link a Pascal program with an FCS resident library, I get two Task Builder error messages complaining about the multiple definitions of the symbols $RLCB and $RQCB. What does this mean?"

The entry points $RLCB and $RQCB are defined by RSX File Control Service (FCS) routines. These routines allocate buffer space when a file is opened. For most RSX programs, this buffer space is allocated from memory reserved in the program section (PSECT) called $$FSR1. On many non-Pascal programs, the user must reserve enough memory in $$FSR1 to handle the maximum number of files that will be open at any one time.

Pascal programs do not use the RSX file storage region $$FSR1. Instead, by defining the entry points $RLCB and $RQCB in the Pascal support library, Pascal programs can Pascal library modules to allocate file buffers. The user then does not need to predict how much memory to set aside for file buffers. If more buffer space is required, the Pascal storage management routines increase the size of the task. When files are closed, the Pascal program can reclaim the buffer space associated with the files and use the space for other purposes.

When Pascal programs are linked with an FCS resident library, the symbols $RLCB and $RQCB are defined in both the Pascal library routines and in the FCS resident library. The FCS library includes the standard FCS routines for allocating file buffers. When you link a Pascal program with this FCS library, the duplicate entry points in the Pascal library cause the two error messages. Pascal itself does not use these entry points, so you can safely ignore the error messages. However, since the standard system buffer allocation routines will be used when files are opened, you must extend the size of the section called $$FSR1 with the EXTSCT option when you build the Pascal task. You should allocate about 540 decimal words for each file to be be opened simultaneously in the Pascal program.

In the most recent version of Pascal for RSX, 1.2F, it is possible to create Pascal resident libraries similar to the FCS resident library. With such libraries, file buffers will be allocated as usual, and you do not have to worry about the special $$FSR1 PSECT.

RSTS/E V7, RT-11 V4

The Pascal-1 documentation describes the operation of Pascal with RSTS/E V6 and RT-11 V2 and V3, and Digital has recently released RSTS/E V7 and RT-11 V4. Both of these new releases will operate with Pascal-1 as distributed.

Here are several points regarding the implementation of the V7 and V4 releases. These points will be covered in detail in future versions of the Pascal-1 manuals.

On RSTS/E V7, the $BUILD program, which controls automatic installation, has one more question in its operating dialogue than does V6. This change, naturally, is not anticipated by the Pascal-1 installation program. Simply answer 'yes' to the installation question about your inspecting the command file, and then run $BUILD manually.

Also, RSTS/E V7 uses the RT-11 V4 Linker, which may cause a memory allocation problem in Pascal-1 revisions prior to V1.2F (see the V1.2 Log on Page 3 of this newsletter). You can run the Linker from RSTS/E V6 to avoid this problem.

The RT-11 V4 Linker contains an apparent bug that causes the program transfer address to be set incorrectly in two instances: when the transfer address comes from a library file, as in Pascal-1; and, when the program uses the overlay facility. The incorrect address causes startup problems on both RSTS/E V7 and RT-11 V4. To resolve the problem, explicitly set the transfer address by using the /T option of the Linker. The transfer address for Pascal-1 programs should be $START.

On RT-11 V4, the Pascal-1 compiler is too large to be used effectively on the extended memory (XM) version of the monitor. In a future V1.2 revision, we will include special versions of the Pascal compiler linked to operate in the XM environment. (The XM compiler will be announced in a future newsletter.) In the meantime, compilations should be run with the SJ or FB monitors, and then linked for the XM monitor. (An easy way to do this is to have both RT11XM.SYS and RT11FB.SYS or RT11SJ.SYS on your system disk, and use the BOOT command to rapidly switch monitors.)

Once compiled, Pascal-1 programs can effectively use the XM environment. The virtual address space allows up to 64KB for programs, and the virtual overlay facility makes program overlays so fast as to be invisible to the user.

Site numbers

Whenever you call us, you likely will be asked for your site number. We use that number to get to your files quickly. Your site number is printed in the headline of all program listings.

You also should include your site number whenever you write us.

We're looking for a few good...

We're looking for Good Stuff. That's any material of interest to users of OMSI Pascal, presented in a well-organized manner.

Your article needn't fit into any formal category; nor does the article necessarily have to involve our products directly. Assume that your reader is an intelligent, literate person (like you), and ask yourself what kind of technical articles you like to read. Then write that way.

Descriptions of company projects are welcome. In fact, we hope to publish many "How I Did It" articles from our users. (We are not, however, looking for Pascal testimonials per se.) Please secure permission from your company for use of your information, and please avoid unseemly corporate chest-beating.

Our official style guide is this: be clear and concise. If you can do that, and if we can read your work (i.e., if you follow normal rules of spelling, grammar, and punctuation, and if you type the thing double-spaced), you stand a much better chance of being published than if you, say, scribble out 30 pages of highly technical jargon in longhand.

Technical style guides are generally available, but most of the rules boil down to your being considerate of the reader: use standard engineering units and other generic terms, and use only common abbreviations. Be sure to provide some background information for readers who may not be familiar with your setup.

You may submit your article in printed form or on a floppy disk or magtape. Include a short descriptive title and a summary of your experience with your article. We will not return any unaccepted article unless you attach a self-addressed envelope with sufficient return postage.

Illustrations (line drawings, graphs, photos, cartoons and the like) are always helpful. Even a rough sketch will do. If we like your article, we'll put our graphic artist to work on finished artwork.

Oregon Software will pay $100 per newsletter page for the articles we publish. We'll pay you upon acceptance.

Reviews

We'll print short reviews of books and articles relating to Pascal. You'll receive a byline, but no payment. Longer reviews of several works or of larger Pascal-related questions may be considered as articles.

Letters, Etc.

Questions? Complaints? Suggestions? Please write. We'll give you any technical help that we can, and we'll print submissions of general interest. We also hope that the newsletter will become a forum for our users, their concerns and interests.

And, if you know of any meetings or activities that might interest other OMSI Pascal users, please notify us of the times and places for inclusion in our Calendar.

Send all letters, queries, Calendar items, reviews and articles to:

Collins Hemingway
Newsletter Editor
Oregon Software
2340 SW Canyon Road
Portland Or 97201

Subscriptions

Every customer supported by Oregon Software will receive one free subscription to the newsletter. The newsletter will be sent to the designated contact person, who (we hope) will ensure that others see it as well.

Because we mean the newsletters to be reference material, we're also sending out a binder in which you can keep the newsletters and other Oregon Software materials. The binder will come in your next regular software shipment.

Additional subscriptions are available for $7.50 a year each. The price includes the binder. Send a check or money order to the address listed below.

OMSI Pascal
NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road, Portland Oregon 97201

OMSI Pascal
NEWSLETTER
OREGON SOFTWARE

Bulk Rate
U.S. Postage
PAID
Portland, Oregon
Permit No. 2079

Oregon Software
2340 SW Canyon Road
Portland, Oregon 97201

Pascal
NEWSLETTER

NUMBER 2 OREGON SOFTWARE MAY 1981

Pascal-2 released; RSX first in line

Pascal-2 is ready.

Oregon Software's second-generation compiler is now being released to RSX users. This high-performance optimizing compiler will operate on all of Digital's RSX operating systems: RSX-11M, RSX-11M Plus, IAS, and VAX/VMS (compatibility mode).

RSTS/E and RT-11 versions of **Pascal–2** are being field-tested and will be released soon.

Pascal–2 is a transportable five-pass compiler that emphasizes conformance to the Pascal standard while generating optimized object code. Written in Pascal, **Pascal–2** will allow programs to be transported between computer systems with few changes.

The **Pascal–2** compiler is larger and compiles more slowly than the **Pascal–1** compiler, but produces code that is shorter and faster. Typical programs are 30 to 40 percent smaller and twice as fast.

Discount Offered

Customers holding **Pascal–1** licenses as of August 31, 1981, may claim a discount equal to 100 percent of the current **Pascal–1** license price toward the purchase of an equivalent **Pascal–2** license. For instance, the current price for a **Pascal–1** U.S. site license is $1950. The same license for **Pascal–2** is $3950. A **Pascal–1** user can therefore purchase the **Pascal–2** license for $2000. International customers will receive a similar discount toward their purchases, based upon international prices.

This discount will apply to all **Pascal–2** orders received by the end of the year.

History of Pascal-2

Pascal–2 grew out of our experience with **Pascal–1**. That product is a one-pass compiler specific to the PDP-11 series, with low-level extensions giving the programmer access to the PDP-11 hardware and operating system. **Pascal–1** is in use at more than 2000 customer sites.

Pascal–1 became operational in 1975. Even as we began to improve it (an effort that continues), we realized that many of our best ideas — whether for code generation or error reporting or what have you — wouldn't work in our single-pass compiler. We began, therefore, to develop a multi-pass compiler. We were determined that this compiler would be the most efficient we knew how to make, that it would track the emerging Pascal standard, and that it would be transportable to different computers. We were spurred, in part, by our own desire to have an efficient, transportable software development tool.

Design of **Pascal–2** began in 1975. The first version of the compiler was completed in 1977. To include better ways of doing things, **Pascal–2** has since been completely rewritten. Twice.

We're proud of the results. **Pascal–2** combines the virtues of a high-level language with the performance of a low-level language, as born out by benchmark tests. The compiler is portable, having already been implemented on a Honeywell machine.

System Described

The **Pascal–2** system consists of the compiler and the Debugger, in binary form; a number of utility programs, in source form; and the documentation, delivered in both printed and machine-readable form. The source versions of the compiler, Debugger, and run-time support library are available at extra cost, with a special license arrangement.

Continued on Page 2

In this issue...

Pascal–2 released . . . . . Page 1
Support policy . . . . . Page 2
Language differences . . . . . Page 3
Conversion to **Pascal–2** . . . . . Page 4
Troubleshooting . . . . . Page 4
  Debugger and RT-11 V4 Linker . . . . . Page 4
  RSX task extensions . . . . . Page 4
  RSX default devices . . . . . Page 4
Pascal calls to RSTS/E system . . . . . Page 5
Making local variables static . . . . . Page 8
Information exchange . . . . . Page 10
Letters . . . . . Page 10
A log of changes to **Pascal–1** V1.2 . . . . . Page 11

The **Pascal–2** system, which is detailed in the accompanying Product Description, has these major features:

• Code is optimized through constant folding, expression targeting, common tail merging, common subexpression elimination, and other techniques.

• Packed structures are implemented, saving space in the representation of data;

• A special "include" lexical directive allows the inclusion of multiple source files;

• A "structured constants" feature allows the initialization of data at compile time rather than at run time, thus saving space and making programming easier;

• Several low-level language extensions are implemented, including direct calls to assembly language or FORTRAN subroutines and direct memory addressing.

• **Pascal–2** will give the source location for nearly 150 types of syntax errors and will detect many more run-time errors than **Pascal–1**.

• Strict interpretation of the emerging Pascal standard eases conversion of programs to other systems. **Pascal–2** extensions can be flagged to indicate non-standard language usage.

The **Pascal–2** system includes a high-level Debugger that helps programmers uncover errors that cannot be caught at compilation time. **Pascal–2** also includes a collection of utility programs, written in Pascal. These utilities, detailed in the Product Description, are designed to extend the capabilities of **Pascal–2** and to lessen the tedium of programming-related work.

The user manual is roughly twice the size of the **Pascal–1** manual and contains many more examples.

Pascal–2, because it uses the same run-time support package as **Pascal–1**, is the logical next step for **Pascal–1** users who want higher performance. **Pascal–1** users, however, will continue to receive support and can look forward to further improvements to that product. In fact, the latest **Pascal–1** update can be ordered now. (The improvements contained in that update are described in the Log of this newsletter.)

Support policy

The license fee for each Pascal system includes one year of software support, including the following:

1) Telephone assistance. We will provide a quick cure to a problem if at all possible.

2) Formal, written response within 30 days to all problems received in writing. Written descriptions are required for complex technical problems to ensure correct diagnosis and repair.

3) A no-cost update to the latest revision of the Pascal product, upon the written request of the user's Designated Contact Person. This is the standard response to bugs that have been fixed. A media fee is charged for media other than magtape and floppies. Shipping via UPS is prepaid; air freight is collect.

4) This Newsletter, which contains status reports on all Oregon Software products, announcements of new versions of software and new products, and various technical articles.

Support, which may be renewed annually, does **not** include consultation in software design.

Customers of an Oregon Software distributor will receive the Newsletter directly from Oregon Software but should contact their distributor for other elements of support.

Site numbers

Whenever you call us, you are likely to be asked for your site number. That number, which we use to locate your files quickly, is printed in the headline of all program listings.

You should also include your site number whenever you write us.

Language differences

The major differences between the language of **Pascal–1** and **Pascal–2** (including the ones most likely to occur) involve the following:

In-line Code: Embedded assembly code is not allowed in **Pascal–2**; such code is not practical in an optimizing compiler. **Pascal–1** in-line code can be rewritten with **Pascal–2** extensions or moved into external MACRO routines.

External Procedures: **Pascal–1** external procedures are defined by the /E compiler switch and are called by the `external` keyword. In **Pascal–2**, the `external` keyword serves both functions. If the body of the `external` procedure appears in the module, then it is available for other modules to use. If the body does not appear in the module, the procedure is assumed to be a reference to another module.

`Array of Char`: `Array of char`, which **Pascal–1** allows for string types, becomes the Pascal standard `packed array [1..n] of char` in **Pascal–2**.

`For` loop control variables: `For` loop controlled variables must be simple variables in **Pascal–2**, local to the routine in which the `for` loop is defined. **Pascal–1** allows any variable to be used.

`Var` Parameter Typing: A variable passed as a `var` parameter must be declared with the same type identifier used to declare the parameter. **Pascal–1** rules are less restrictive.

Other differences are summarized below. The Conversion Guide of the **Pascal–2** manual includes a complete list of all differences and the programming changes required to accommodate them.

Improvements

Packing of structures is implemented; a record allocated on the heap by `new(ptr, tag1, tag2, ..., tagn)` is allocated the exact amount of storage required for the variants specified; the standard procedures `pack` and `unpack` are implemented; the procedures `read` and `write` apply to all file types; structured constants can be defined; the `loophole` function provides a method of bypassing type compatibility rules; the `%include` lexical directive simplifies the use of header files or multiple source files; maximum set size is increased from 64 to 256 elements.

Error Checking

Pascal–2 does the following kinds of error checking that **Pascal–1** does not: assignments of out-of-range values to subrange variables; dereference of a nil pointer; many uses of an uninitialized variable; any attempt to assign a value to a `for` loop variable within the loop; the passing of a component of a packed structure as a `var` parameter to a procedure; the redefinition of a name in a scope.

Changes to Extensions

The `fortran` directive has been renamed to `nonpascal`; `origin` variables can have addresses only in the system space (less than 1000B) or in the I/O space (greater than or equal to 160000B); single-character names are not allowed as the file name and default arguments for `reset` and `rewrite`; the `ref` function replaces the "`@`" as an address operator; the `otherwise` clause in a `case` statement serves the same function as the `else` clause in **Pascal–1** (`otherwise` will apparently become a standard Pascal extension).

Removal of Extensions

These extensions have been removed: the `exit` statement (it can be replaced with a `goto` statement); the **Pascal–1** pre-declared type `alfa`; the pre-defined function `float` (`R:=I` is equivalent to `R:=float(I)`); the pre-defined functions `log` and `exp10` (they can be defined in terms of `ln` and `exp`); several substitute characters allowed in some early versions of **Pascal–1**; the non-standard comment brackets "/*" and "*/" (the PASMAT formatter will replace these brackets with standard brackets).

Changes to or Clarifications of the Standard

These changes result from **Pascal–2**'s conformance to the latest International Standards Organization draft standard for the language:

A full procedure heading is now required for any procedure or function to be passed as a parameter; strings cannot cross lines; `input` and `output` are predefined as global variables in every program; the "`@`" is an alternate character for the pointer symbol.

Changes in Implementation Definitions

ASCII control characters, including tabs, are not allowed in string constants; characters are seven bits, not eight.

Conversion to Pascal-2

Pascal–1 users should take heart in the knowledge that converting their programs to **Pascal–2** is, in most cases, a straightforward proposition.

Conversion normally requires nothing more than the changing of embedded switches to the equivalent switches in **Pascal–2**, the compilation of the modified program under **Pascal–2**, and the correction of any errors that appear as a result of **Pascal–2**'s improved error checking.

To aid conversion, the **Pascal–2** manual has a Conversion Guide, which describes the process step by step. The Conversion Guide explains the programming changes required to accommodate each of the differences between **Pascal–2** and **Pascal–1**.

Among some of the more important changes, **Pascal–2**'s improved error checking may require the reworking of some **Pascal–1** programs. The resulting programs, however, should be more reliable than before. The Conversion Guide contains a section on "Likely Error Messages and Countermeasures" as a reference.

You also may have to substitute **Pascal–2**'s standard symbols for some non-standard **Pascal–1** symbols, but these changes are easy to make.

Included in the **Pascal–2** package is the `CONVRS` utility. `CONVRS` scans a program to produce a list of potential conversion problem areas, with the file name, line number, and text of the line where the problem was noted. `CONVRS` will flag embedded switches, external routine definitions, in-line code, use of variant records without tag fields, and the use of **`origin`** — those areas likely to require the most work. `CONVRS` is described in detail in the Conversion Guide.

Conversion of external procedures is forthright. The key is to be able to convert the main program to **Pascal–2**. If you can't, you can convert only procedures that do not have global references. You can convert external modules one at a time, and you can use the /PASCAL1 compiler switch to call routines that are difficult to convert. (All these steps are explained in the Conversion Guide.)

You must rework any assembly code in a program (this is often possible with the low-level extensions of **Pascal–2**); or code an external routine in assembly to do the function required (this is usually easy to do with the PASMAC utility macros provided with **Pascal–2**); or use the `/PASCAL1` compiler switch to call these routines.

A few **`origin`** variables may require the use of the **Pascal–2** **`loophole`** function (explained in the Conversion Guide).

Before conversion, the **Pascal–2** library must be installed in place of the **Pascal–1** library. The **Pascal–2** library contains all of the routines in the **Pascal–1** library, plus additional routines used by **Pascal–2**. The **Pascal–2** library can be used by both compilers.

One final note: Some conversions might be difficult. You must understand any tricky parts that **you** have put into programs, and you must read the **Pascal–2** Language Specification before attempting any conversion.

Technical Troubleshooting Debugger and RT-11 V4

A bug in the RT-11 V4 Linker incorrectly sets the starting address of overlaid programs that have a transfer address in a library, as in Pascal-1 programs. The user must explicitly use the /T switch in the first Link command, giving the symbol `$START` as the answer to the `"Transfer symbol?"` question.

RSX default devices wrong

In some cases the default system device would be incorrect on multi-disk systems in which the user's default device was different from the device from which the Pascal task was installed. The solution is to specify `SY0:` explicitly in the default filename field for installed tasks.

RSX task extension

The global symbol `$NOEXT` may be patched to be a `NOP` to prevent Pascal programs from automatically asking RSX for more memory when no space is available on the free list. The patch is applied with the Task builder option:

```
TKB> GBLPAT=progname:$NOEXT:240
```

This patch may be needed with resident overlays, when a Pascal task is mapping to shared common libraries, or when other uses are being made of `PLAS` directives. This patch prevents Pascal programs from extending into virtual address spaces being used for other things. If this patch is used, dynamic memory must be allocated explicitly in `$$HEAP`.

Pascal system calls to RSTS/E

The following program demonstrates the implementation of system calls to the RSTS/E operating system. The program shows the opening of a file in record update mode on RSTS/E. Note that the data file must be created and pre-extended before this update program is run. This program also shows the internal structure of the RSTS/E file channel for **Pascal-2**. (The file channel structure for **Pascal-1** is not the same because the two compilers allocate storage in different ways.)

In-line MACRO code is not allowed in Pascal-2 source text. Note how the **emt** procedure and variable definitions using **origin** allow the **Pascal-2** programmer to accomplish the same results as with in-line code.

```
program UpdateExample(input,output);

label 99;

  {Oregon Software reserves the right to change the internal structure
   of file variables in future revisions of Pascal-2.}

const
  DataFile = 'SY:DATA.DAT';

type
  Byte = 0..255;
  Rad50 = 0..65535;
  StatusBits = (MustWrite,Spanned,SeekOk,DoRead,DoWrite,PendingEof,Temp,Text,
                NotUsed,Go,Defined,Wait,Odt,Dirty,EolnFlag,EofFlag);
  DeviceBits = (NFS,ReadLock,WriteLock,Position,Modifiers,Force,Keyboard,
                Random);

  {define a block}
  FileRecord = packed array [0..511] of Byte;

  {description of internal file data block}
  FileBlock = packed record
  RecordPointer: ↑FileRecord; {pointer to current block}
  Status: set of StatusBits;
  DeviceModifier: integer;
  WaitTime: integer;
  Device: integer;
  Unit: Byte;
  UnitFlag: -128..127;        {actual values are -1 and 0}
  Programmer, Project: Byte;
  Name1,Name2,Extension: Rad50;
  HandlerIndex: Byte;
  DeviceType: set of DeviceBits;
  BufferSize: integer;
  BufferAddress: integer;
  EndOfData: integer;
  RecordSize: integer;
  CurrentBlock: integer;
  Channel: Byte;
  HighBlock: Byte;
  end;
```

Continued on Page 6

```
FileBlockPointer = ↑FileBlock;

FileType = file of record
{Define your record structure here}
  I : integer;
  R : real;
  B : boolean;
  C : char;
end;

var
  F: FileType;
  RecordNumber: integer;
  WeHaveTheRecord : boolean;

  procedure Error(code : integer);
  begin
    writeln('Fatal RSTS Error #',code:1);
    goto 99;
  end;

  procedure Sleep(Seconds: integer);
  var
    XRB origin 442B: array [0..6] of integer;
  begin
    XRB[0] := Seconds;          {put sleep time in XRB}
    emt(255);                   {RSTS EMT}
    emt(8);                     {and sleep}
  end;

  function Channel(F: FileBlockPointer): integer;
  {return RSTS/E channel number from file variable}
  begin
  Channel := F↑.Channel div 2;
  end;

  procedure ForceRead(F: FileBlockPointer);
  {force buffer refresh on next seek}
  begin
    F↑.HighBlock := 255;
    F↑.CurrentBlock := -1;      {set to non-existent block}
  end;

  procedure Lock(F : FileBlockPointer);
  var
    XRB origin 442B: array [0..6] of integer;
    I: integer;
  begin
    for I := 2 to 6 do XRB[I] := 0;
    XRB[0] := 2;                {explicit lock}
    XRB[1] := F↑.CurrentBlock;  {record number}
    XRB[3] := Channel(F) * 2;   {get correct channel}
    emt(255);                   {RSTS/E .SPEC}
    emt(12);
  end;
```

Continued on Page 7

```
procedure Unlock(F : FileBlockPointer);
var
  XRB origin 442B: array [0..6] of integer;
  I: integer;
begin
  {when XRB[1] = 0 RSTS will release all locks on the file}
  for I := 0 to 6 do XRB[I] := 0;
  XRB[1] := F↑.CurrentBlock;  {record number}
  XRB[3] := Channel(F) * 2;   {channel}
  emt(255);                   {RSTS/E .SPEC}
  emt(12);
end;

procedure GetRecord(var F: FileType;
                    RecNum: integer;
                    var UserHasRecord : boolean);
const
  SleepTime = 5;              {The wait until next try for locked record}
  MaxTrys = 5;                {Limit tries to avoid deadly embrace}
var
  ErrCode origin 402B: integer;{RSTS/E IOSTS (I/O status)}
  NumberOfTrys : integer;
  FirstTime: boolean;
begin
  FirstTime := true;
  NumberOfTrys := 0;
  UserHasRecord := false;
  repeat
    ForceRead(loophole(FileBlockPointer,F));
    seek(F, RecNum);
    if ErrCode <> 0 then      {an error of some kind}
      if ErrCode = 19 then begin{record locked by other user}
        if FirstTime then begin
          writeln('Waiting on a locked record');
          FirstTime := false;
          end;
        Sleep(SleepTime);     {and relax}
        NumberOfTrys := NumberOfTrys + 1;
        end
      else Error(ErrCode);       {other errors are fatal}
        until (ErrCode = 0) or (NumberOfTrys = MaxTrys);
        if ErrCode = 0 then begin{success}
          UserHasRecord := true;
          Lock(loophole(FileBlockPointer, F));{claim the entire block}
          end;
        end;

      procedure ReleaseRecord(var F: FileType);
      begin
        Unlock(loophole(FileBlockPointer,F));{give back the block}
      end;
```

Continued on Page 8

```
begin
  reset(F,DataFile,'/seek/go/mode:1');{update mode}
  repeat
    write('Record Number > ');
    readln(RecordNumber);
    if RecordNumber > 0 then begin
      GetRecord(F, RecordNumber,WeHaveTheRecord);
      if WeHaveTheRecord then writeln('You have the record');

      {process record in F↑}

      ReleaseRecord(F);
      end;
  until RecordNumber = 0;
  99:
end.
```

Making local variables static

By SERGIO ANTOY

(Mr. Antoy is with the Laboratorio per la Matematica Applicata, Consiglio Nazionale delle Ricerche, in Genova, Italy. His installation has the RSX-11M operating system V3, **Pascal-1** compiler V1.1G and TECO editor V24.)

The value of a variable local to a Pascal procedure or function is defined only during the execution of the block to which the variable belongs. A subroutine, through its variables, cannot hold values from one call to the next.

While not much of a problem when every subroutine is part of a main program, this characteristic sometimes becomes a flaw when a subroutine, which will later be linked to a main program, is compiled separately, something that Pascal compilers often allow.

One way to write separate Pascal modules that hold values between successive calls is to use a processor consisting of a RSX indirect command file. The processor accepts for input the source assembly code generated by the **Pascal-1** compiler and produces for output the same code slightly modified by the TECO editor. Normally, global-level (outmost) variables of the module would be mapped over the global-level variables in the main program. The modification causes these variables to instead behave as static variables, i.e., variables unrelated to the main program and whose value is not lost at the subroutine termination.

The processor is listed below.

```
.ENABLE SUBSTITUTION
.ASKS $FIL MODULE NAME
.XQT PAS '$FIL'='$FIL'/S
.ASKS $IZE STATIC SIZE
.SETS $NUL ""
.OPEN TECO.TEC
.DATA @I/ .TITLE  '$FIL'
.SETS $TTC "$TTC"
.IF $IZE = $NUL .GOTO 1
.DATA '$TTC':     .BLKW    '$IZE'
.GOTO 2
.1: .ASKS $TTC ENTRY POINT
.DATA     .GLOBL  '$TTC'
.2: .ENABLE DATA
/
@EB/'$FIL'.MAC/
!LOOP! AZ"N
<@S/(5)/; @FR/+'$TTC'/>
<@S/%5/; @FR/#'$TTC'/>
0,ZPW 0,ZK
.DISABLE DATA
.DISABLE SUBSTITUTION
.DATA @0!LOOP!
.DATA EX
.CLOSE
.WAIT PAS
TEC
.ENABLE SUBSTITUTION
.XQT MAC '$FIL'='$FIL'/EN:LC
PIP TECO.TEC;*/DE
```

Consider an example.

The function **Lapse** returns the time elapsed between two consecutive calls. The value returned by the first call is meaningless. The time is measured by the predeclared function **time**. The Pascal module implementing such a function is listed below. Note that the value of the variable **Previous** cannot be lost from one call to the next.

```
{$E+}

var Previous : real;      {to be static}

function Lapse : real;
var Present : real;
begin
  Present   := time;      {predefined}
  Lapse     := Present - Previous;
  Previous  := Present    {store value}
end;
```

The processor performs the following steps.

1) The user is requested to supply the name of the module, for example LAPSE. The extension must be .PAS and should not be supplied. The module is compiled as usual, and the file LAPSE.MAC is generated by the compiler.

2) The user is requested to supply the size, in 16-bit words, of the storage devoted to the static variables. In this example, two words are enough.

3) The file TECO.TEC is generated and is used as a command file by the TECO editor for editing the file LAPSE.MAC. A new version of this file is then generated.

4) The edited version of the file LAPSE.MAC is assembled as usual, and the file LAPSE.OBJ is generated.

5) The file TECO.TEC is deleted.

TECO.TEC depends on the answers given by the user to the processor and exists in the system only for the short time in which it is needed. For this discussed example, the file contains:

```
@I/          .TITLE  LAPSE
$TTC:        .BLKW   2
/
@EB/LAPSE.MAC/
!LOOP! AZ"N
<@S/(5)/; @FR/+$TTC/>
<@S/%5/; @FR/#$TTC/>
0,ZPW 0,ZK
@O!LOOP! '
EX
```

The TECO editor modifies LAPSE.MAC in two ways:

1) The following two statements are inserted:

```
            .TITLE   LAPSE
$TTC:       .BLKW    2
```

The first line changes the default .MAIN. title of the module so that several modules can be inserted in a library. The second line reserves the storage for the static variables. The name $TTC has been arbitrarily chosen for addressing such a storage.

2) Every reference to a global-level variable is modified. The addressing of global-level variables is normally done through register 5 in either index mode or register mode. The two lines of the TECO command file enclosed in angle brackets show how each kind of reference to register 5 is modified to switch the address of a variable from the global-level storage to the static storage.

This processor allows a variation to separately compile subroutines sharing static variables. When giving the size of the static storage, the user can enter a null answer, i.e., a carriage return only. That produces two effects: (a) no storage is reserved inside the module being compiled; and (b) the processor directs a new question to the user. The storage for the statuc variable has to be defined by the user in a separate module. The symbolic name chosen by the user for addressing the static storage has to be defined as global in its module. The user then gives this name to the processor upon request and a suitable file TECO.TEC is generated.

Users can tailor the processor to particular needs. For example an .IDENT clause can also be added to a module, and each line of the source assembly code modified by the processor can be flagged with an audit trail comment. To do this, the processor need only generate the following TECO.TEC file:

```
@I/     .TITLE  LAPSE          ;*** NEW ***
        .IDENT  ↑X2F↑          ;*** NEW ***
$TTC:   .BLKW   2              ;*** NEW ***
/
@EB/LAPSE.MAC/
!LOOP! AZ"N
<@S/(5)/; @FR/+$TTC/ L2R@I/ ;*** NEW ***/>
<@S/%5/; @FR/#$TTC/ L2R@I/  ;*** NEW ***/>
0,ZPW 0,ZK
@O!LOOP! '
EX
```

This processor has been used in a wide variety of applications and particularly in the implementation of a graphic library. The static variables, such as current position of the plotting pen, scale and rotation factors, etc., are inclusive

Continued on Page 10

of scalar, real and pointer types both used individually and structured in arrays. The processor proved to be a reliable and useful tool in accomplishing the task.

Editor's Note: This method, or one similar, is needed to make local variables behave as static variables in a **Pascal–1** program. The same effect can be achieved in **Pascal–2** programs by means of the `/OWN` compilation switch.

Letters

From: D. B. Cooper

Subject: Late-Breaking Pascal Developments

Dateline Berkeley: Doug Cooper receives priority mail from a uniformed government employee — new Pascal Standard proposal available ... details soon ...

Dateline Japan: Japanese suggestion that "the work of three years be thrown away and a new draft written from scratch" is coolly received by Anthony Addyman, draft standard's author ... That's telling 'em, Tony.

Dateline France: ISO/TC97/SC5/WG4 report just in ... asks that we "spare a thought for our colleagues in France [because] to achieve an ISO standard an equivalent French text is required" ... Thus please "only propose necessary changes" to proposed standard ... Quel fromage, hey?

Dateline Tasmania: Arthur Sale proposes that **`repeat`** ... **`until`** statement be dropped from Pascal ... says change "will increase the probability of correct programs" ... claims that "this is no joke" ... Go to it, Art.

Dateline ISO/TC 97/SC 5 N959 (X3J9/81-003) DP7185: Conformant arrays, presumed dead, are resurrected in new draft proposal for "two levels of compliance–levels 0 and 1" ... Level 0 does not include conformant array parameters, Level 1 does ... Guess we should have used silver bullets ... "powers of 2" level naming (0, 1, 2, 4, 8 ...) expected soon.

"What Does It Mean" contest: "For benefit of those with access to the papers WG4, the 2nd Draft Proposal has been change barred in those places where the text differs from that produced by applying the proposed changes of WG4 N9 to WG4 N4" ... Say, aren't those the markings on the Enterprise?

New Concepts For You To Learn: Schema-arrays ... meta-identifiers ... apostrophe-images ... implementability ... the "Detergent Molecule View of Parameters".

Updates when available ... film at 11.

Information exchange

We often get inquiries about certain kinds of programming applications, and we often know a customer who is experienced in that area.

Because our customer names are confidential, however, we cannot steer someone toward a particular user without permission. And that's hard to get with about 2000 customers around the world.

One purpose of the Oregon Software Newsletter is to serve as an information exchange among our users. If you have any questions on a topic, you can write us and we'll publish your query. Any reader who has information can then contact you directly.

Similarly, if you are involved in a technical project that might be of interest to other users, you can write us and briefly describe the application. We'll publish the description so that any user who desires can contact you directly. Limit of about five lines. Something like:

> Real-time data collection and process control of sewage plants using Pascal and RSX-11M, John Doe Co., [address, phone number].

You also may wish to submit a technical article describing your application. We've had recent inquiries on real-time process control, database management, and the accessing of RMS file structures or DECnet from Pascal. Any of these topics is worth a newsletter article. We pay $100 per published page for articles.

(And, please, we're looking for a technical rather than marketing orientation in these items.)

About those binders ...

Our promise to deliver has put us in a bind.

Subscriptions to this newsletter include a three-ring binder, which we had hoped to deliver by now. We were not, however, satisfied with the printing quality of the first shipment of binders, and we are now preparing another batch.

When finished, the binders will be placed in the next regular software shipment for customers. The binders will be mailed to non-customer subscribers.

Each customer receives one free subscription, which goes to the Designated Contact Person. Additional subscriptions are available for $7.50 a year each U.S. and $10 a year international.

The Log: Pascal-1 V1.2

Oregon Software's first Pascal Newsletter described the significant changes in each version of **Pascal–1** V1.2 from its initial release, V1.2A, in December 1979 through V1.2F in November 1980. This issue describes the significant changes in the software in the two subseqent updates. **Pascal–1** V1.2H can be ordered as of June 1.

To use this log, first determine what release of V1.2 you have. (The release number is printed on the headline of all program listings.) Then review the log to determine the changes made since the release of your version. If the changes are of particular importance to your application, your Designated Contact Person should request an update, in writing. The DCP is usually the senior programmer in charge of software. You will receive the latest version of the software.

This log describes only the most important changes made to V1.2. A copy of the complete log is available upon request. Future issues of the newsletter will continue to outline improvements to Oregon Software's Pascal products.

The first section of this log describes changes applicable to **Pascal–1** for all operating systems. The other sections describe **Pascal–1** changes specific to each operating system. The version listed is the one in which the change first appears; succeeding versions also include the change.

Changes to all systems

V1.2G corrected these problems:

"Dispose" of short records

When disposing of a single-word item, the **dispose** procedure cleared the next word, possibly overwriting valid data.

Bounds checking of scalar indices

The compiler generated incorrect bounds checks for array accesses indexed by a user-defined scalar type. You can circumvent this problem by disabling bounds checking with {$A-}. This problem appeared only in V1.2F.

Tab-stop alignment

The source program in compilation listings began one space to the right of a tab stop, causing improper formatting of program listings that contained tab characters.

V1.2H corrected these problems:

IMP branch instructions

The IMP branch/jump resolver would in some cases generate a branch instruction with a destination which could not be reached, producing addressing errors from the assembler. IMP has been corrected, and users can correct their version of IMP by changing two occurrences of <=127 to <127 in the source.

Size of file variables

Compiler now allocates 2 bytes (not 14) for file variables of type **text**. (All other file variables were being correctly allocated.)

Dynamic memory allocation

The heap management routine was modified to prevent memory fragmentation when blocks of similar size are allocated and deallocated. Also, when variables are disposed, the released memory is filled with negative ones (-1) to help catch dangling pointers and unitialized variables. (Programs that previously ran may now fail with "Odd Address Trap" errors involving pointers, indicating latent bugs.)

Blank fields in variant records

Compiler now allows an empty field list in variant record contexts.

Negative zero

Floating-point simulator modified to prevent a negative zero in the case of 0.0 - 0.0. The problem only occurred on machines with no floating-point hardware.

$E switch

The $E embedded switch now turns on and off correctly.

Round function

I:=Round(F(1/2)) attempted to float the left-hand side if F was a **real** function.

Error processing

Array (1..10) of <type> now gives an error message rather than a trap.

Continued on Page 12

Array index checking

Array index checking is no longer suppressed if the index type is not a subrange.

Single-precision constant conversion

Compiler now rounds, rather than truncates, when converting single-precision real constants; the result is more accurate single-precision representations.

/X made known to Debugger

The compiler used to write the initial symbol table before checking for the /X switch, making it impossible for the Debugger to detect whether the program was compiled with the /X switch.

Double-precision logarithm

The double-precision logarithm function contained an SXT instruction not guarded by a conditional assembly. This instruction was replaced by non-EIS code that performs the same function.

Changes to RSTS/E

V1.2G corrected this problem:

FIS error handling.

Floating Instruction Set (FIS) error traps were incorrectly handled, causing unpredictable and fatal error messages.

V1.2H corrected these problems:

/DET CCL switch

The /DETACH switch, when used in a CCL command that executed a Pascal run-time system, sometimes failed.

Privileged programs

On systems with small user swap maximum sizes, privileged programs were not allowed to use the larger maximum normally granted to privileged users.

"Eof" set incorrectly with "seek"/"put"

If, while writing a file sequentially, a **seek** repositioned the file pointer within the current buffer, subsequent **put** operations would not set **eof** when the previous end of data was reached.

Long records

Files of long records would, in some cases, fail to read the first record after a **reset**.

Changes to RT-11

V1.2G corrected these problems:

Fast assembler MAC

Some small programs produced incorrect .OBJ code, and global symbols had incorrect flag bits that caused "?Null library" messages from the LIBR librarian.

Debugger, Profiler exit

The Debugger and Profiler terminated program execution on a Control-C interrupt rather than regaining control of the program. Two consecutive Control-C characters now return control to the Debugger or Profiler (RT-11 V3 or later only).

V1.2H corrected these problems:

"Eof" set incorrectly in "seek"/"put"

If, while writing a file sequentially, a **seek** repositioned the file pointer within the current buffer, subsequent **put** operations would not set **eof** when the previous end of data was reached.

/D switch in PCL

The PCL program, when using only the /D switch, caused the Debugger symbol table file to be sent to the terminal, rather than to the disk. A user may modify PCL.PAS and recompile.

/NFS file switch added

The /NFS switch is used in **reset** and **rewrite** parameters to indicate that non-file-structured access to a device is desired. An error is returned if a direct-access device is opened without including this switch. (The previous method of specifying NFS access still works.)

Problem with fourth "reset"/"rewrite" parameter

Use of a default extension sometimes caused the value returned in the fourth parameter to incorrectly indicate that a **reset/rewrite** completed without error.

Continued on Page 13

Problem reading long records

Files of long records (greater than 512 bytes) would, in some cases, fail to read the first record after a **reset**.

SJ monitor reset on termination

Pascal–1 programs operating on the SJ single-job monitor were not causing a system reset instruction upon exit, as required.

Integer arithmetic on EIS, SIM

In complicated expressions involving both **integer** and **real** values, the **integer** part of the calculation was sometimes corrupted by the floating-point simulation routines.

Duplicate "dispose" under XM monitor

Under the RT-11 XM monitor, some programs incorrectly gave the message "Duplicate Dispose()" when closing files via **close** or upon program termination.

Changes to RSX

V1.2H corrected these problems:

Previous LUN assignments on file opens

If a file was opened on some device, say TI:, then closed, then another file was opened on a different device, the LUN assignment performed by FCS on the initial file open would cause the second file to be opened on TI: if no explicit device was given for that file. The support library now reassigns the LUN to SY0: after the file is closed.

"Rewrite" of files

Rewrite now explicitly truncates the file to handle problems in which an existing file is opened with an explicit version number. The preallocated blocks remain allocated, but the logical end of file is before the first record.

Closing of files

Changes have been made in the support library's closing of files while cleaning up during the termination of a Pascal task.

Unused .GLOBL references

Compiler now prevents the blind generation of .GLOBL statements for all programs. The .GLOBL statements were not required for RSX; their presence made it impossible to place compiled Pascal external procedures in resident libraries.

Exit with status added to compiler

The compiler now sets a termination status to permit detection of bad compilations.

Bad symbol table generation

In some forward type references, the symbol table file was corrupted when the target record was in a different disk block than the current record.

Integer overflow now fatal

An integer overflow during a signed multiply is now fatal rather than a warning. Also, the message for overflow during a **trunc** or **round** operation is now "Trunc/Round Overflow", and the error is now fatal.

Rounding on big buffer file opens

The Pascal open routine now rounds up to the next multiple of 512 to prevent the possibility of errors when the /BUFF switch is used to specify buffers greater than 512 bytes when files are opened. (Some RSX systems and emulators require the buffer size to be a multiple of 512 bytes.)

/APD switch sets "eof"

The /APD switch would sometimes position a file past the last record, but it would not attempt to read the next record. This might prevent **eof** from being set when /APD was used.

Control-C in Debugger

Control-C problems are still being reported in the use of the Debugger. The problem may be related to whether the full-duplex or half-duplex terminal driver is being used. The Debugger interface was changed to explicitly detach and reattach with AST in a slightly different way to try to resolve the problem.

Continued on Page 14

Size parameter in "rewrite"

A change in V1.2C caused the size parameter in a `rewrite` to be ignored. The problem was fixed. The /SIZE switch was not affected.

Shipping of orders

All prepaid rush orders are shipped within three business days of receipt, air freight. All regular orders are shipped within 20 business days of receipt, via UPS.

All maintenance releases (updates) are shipped within 20 to 30 business days. Updates to the latest revision of the Pascal product are available at no cost upon the written request of a user's Designated Contact Person. A media fee is charged for media other than magtape and floppies. Shipping via UPS is prepaid; air freight is collect.

Printed in USA

9240 SW Canyon Road, Portland, Oregon 97201

Pascal
NEWSLETTER

NUMBER 3 OREGON SOFTWARE NOVEMBER 1981

Memory allocation for Pascal programs

Some Pascal users report that (apparently) small programs will run out of memory on RSX. Often sufficient memory exists, but for various reasons it is not available. To make better use of memory, users should understand how it is allocated and used by Pascal programs.

Background on RSX

Task size limits: Each separate task can address and use at most 32K words (or 64K bytes) of memory, even though the system may have much more main memory. This limit is caused by the PDP-11 16-bit address range. Overlays will not increase this address space, but may allow reuse of some of the space for different purposes.

Program organization: The parts of a Pascal program are described under "Run-Time Organization" in the Programmer's Guide of the Pascal–1 or Pascal–2 User Manuals: task header, program code, constants, global variables, tables, stack, and heap. Refer to the run-time section as needed while reading this article.

Static and dynamic allocation: Much of a task's memory allocation is static (it does not change over time), set by the Task Builder at task-build time; some memory is dynamic, allocated by the Pascal program at run-time. There are two kinds of dynamic memory allocation: the "stack" and the "heap".

Dynamic uses of memory — the stack and the heap: Each time a procedure is called, stack space is reserved for the variables declared in that procedure; when the procedure terminates, or is exited by a **goto**, that stack space is released. When a file is opened, heap space is reserved for the file buffer and file descriptor; each use of the **new** procedure also reserves space on the heap. **Close** and **dispose** release heap space used for files and for **new**, respectively.

Basic Task Builder operations

Compilation of a program produces a relocatable object module (or modules, if there are external routines). The Task Builder combines the program module(s) with object modules extracted from the Pascal support library (PASLIB) and the system library (SYSLIB). All the object modules contain one or more program sections called PSECTs. The Task Builder sorts the PSECTs into alphabetical order, allocating space for the program header, code, constants, and tables of the Pascal program. Pascal–2 tasks contain a PSECT for the global-level variables associated with the program. Pascal–1 tasks do not; instead, global-level variables are allocated when the task is initialized.

The Task Builder also allocates a small stack space in the task image. Pascal programs, however, require a large stack space to hold the local variables associated with each procedure, of which there can be many. The small stack space set up by the Task Builder is used by Pascal tasks only for initialization of the program at run-time.

All Pascal tasks contain a PSECT called `$$HEAP`. Normally, the Task Builder does not allocate space to the `$$HEAP` PSECT. In this case, the Pascal initialization routine uses the `EXTK$` (extend task) system directive to expand the task by 2K words; this space is used for the heap and the stack. If the `$$HEAP` PSECT has been extended via the `EXTSCT` Task Builder directive, described below, `$$HEAP` is used instead for the stack and the heap. In Pascal–1 programs, the global-level variables are then allocated (also by task extension or from `$$HEAP`).

Continued on Page 2

In this issue...

RSX memory allocation . . . . . Page 1
Reviews . . . . . Page 3
Pascal–2 released for RSTS, RT-11 . . . . . Page 4
Letters . . . . . Page 5
Information exchange . . . . . Page 7
'For' loop restriction . . . . . Page 7
Trouble with trouble reports . . . . . Page 8
Pascal for text formatting . . . . . Page 9
The Log: Pascal-1 and Pascal-2 . . . . . Page 13
RT-11 disk space problems . . . . . Page 16
RT-11 Linker problem (again) . . . . . Page 16
TSX and TSX-Plus . . . . . Page 16
Additions, corrections to manuals . . . . . Page 17

Stack and heap allocation

The stack starts at the top of the initial allocation and expands down toward low memory. A "stack pointer" points to the bottom of the active stack; as the stack grows, the stack pointer (an address) becomes smaller. The stack cannot grow below the lower limit of the initial allocation.

Demands for heap memory (the use of **new** or the opening of files) cause the task to be extended (in 1K-word chunks). The additional memory is allocated to the heap, up to the system maximum (usually 32K). When the task can be extended no farther, the heap then uses any space available at the *bottom* of the initial allocation, that is, below the stack. The heap will continue to grow upward. The stack will continue to grow downward. If the heap and stack collide, the task is "out of memory".

Note that task extension creates additional space for the heap, but not for the stack. At most, the stack can use all of the space initially allocated (either the default of 2K words, or the amount explicitly allocated at task-build time). Therefore, a program that makes heavy use of the stack (for local variables or recursion) will require explicit allocation for the `$$HEAP` PSECT at task-build time.

RSX requirements for task extension

As you can see, the ability to extend the task at run-time is crucial for automatic allocation of the heap and stack. Each of the following points should be checked to allow task extension.

The `EXTK$` directive should be included in the RSX system during SYSGEN. `EXTK$` is used by other system software, so it is usually present; it may not, however, be available on RSX-11S systems.

For a task to be extended, it must be checkpointable (which is why the `/CP` switch should be used when in the building of Pascal tasks). Also, the checkpoint file must be active. The privileged MCR command `ACS` is used to allocate checkpoint files.

Additionally, the system manager can control the maximum amount of memory that a task is allowed to add to its original size. If programs are running out of memory before reaching the 32K-word limit, use the `SET /MAXEXT` command to determine the maximum extension allowed: the limit may be less than 32K words.

Diagnosing errors

The error message, "Not enough memory. Make task checkpointable or extend $$HEAP", means that not enough stack, or, for Pascal–1, to allocate the global-level variables. This error occurs at initialization; the program has not yet started. If you've already made the task checkpointable or extended $$HEAP, the problem is that the maximum task extension is set too low, `EXTK$` is not enabled, or checkpointing is not enabled.

The error message "Stack overflow" means the stack is too small to hold all the local variables. A procedure containing a local array of 4500 integers, for instance, will produce a "stack overflow" even when plenty of memory is available. The array requires 4500 words on the stack, compared to the default size of 2K words (2048 words). Task-building the program with the `$$HEAP` PSECT extended to 5K words will solve the problem.

The error message "Not enough memory" means that requests for memory from the heap (**new** for example) exceed the available memory. If a program runs out of memory because of heap accesses, check to see whether you can close any files, thus freeing up the buffer space and control blocks they are using.

If task extension isn't possible

If a task cannot be checkpointed, or if the extend-task directive is not available, additional space cannot be automatically allocated. In this case, you must estimate the amount of memory required for the combined heap and stack. Take into account the number of open files (at about 600 bytes per file), the size of local variables, and, for Pascal–1, space for global variables. See the section "Storage Allocation" in the Pascal–1 or Pascal–2 Programmer's Guides. For instance, a procedure with a local array of 8000 integers will require 8000 words (16,000 bytes) of stack space for the array. It's better to overestimate (so your program will run), but avoid using too much memory, because a non-checkpointable task occupies physical memory until it finishes.

Then, explicitly allocate the estimated space for `$$HEAP` with the Task Builder option `EXTSCT=$$HEAP:num`, where `num` is the desired size of the PSECT. (`Num` must be expressed in octal bytes.)

To complicate the issue ...

Sometimes, you may want to prevent a Pascal task from growing, yet still want it to be checkpointable. An example is a collection of Pascal tasks communicating with one other. If each task were allowed to grow to 32K words, the loading of one task could cause another task to be swapped

out. Another example involves programs making special use of virtual memory via the PLAS directives. Patching the global symbol $NOEXT as a Task Builder option will prevent Pascal tasks from requesting more memory, yet keep them checkpointable. The format is shown in this example with a program called PROG:

```
GBLPAT=PROG:$NOEXT:240
```

Since this patch prevents task extension, you must explicitly allocate space in $$HEAP, as described above.

Overlays

Even after using all the methods described above, the program still may run out of memory. The only alternative is to overlay the program.

If you must use overlays, inspect the file PAS.ODL distributed with the Pascal–1 and Pascal–2 systems for RSX. This overlay description file shows how to overlay the modules in the Pascal support library and the system library, thus saving room.

See also the "Overlays" section of the Programmer's Guide in the User's Manual.

Changes in name, address

"Request to Amend" forms are now being sent with all new software shipments and updates.

Our license agreements require a customer to complete the "Request to Amend" form whenever the name or address of the customer changes or a new Designated Contact Person is named. The appropriate changes should be noted on the form, and an authorized official should sign the amendment.

Customers should mail the amendment to our sales department. We'll return a copy signed by one of our authorized officials. Completion of the form constitutes a legal change in the license agreement.

Reviews

(**Introduction to Pascal**, Rodnay Zaks, Sybex, Inc., Berkeley, California, 1981. 422 pages, $14.95 softcover; no hardcover. Available through Oregon Software and elsewhere.)

Whether you're new to programming or simply new to Pascal, this book is without question the best organized and the most clearly written of the many introductory Pascal books now available.

Introduction to Pascal combines clear discussions of Pascal with the basics of programming techniques — a double plus for beginners. The book's strong organization helps intermediate programmers find specific information quickly and easily. Advanced programmers will also find the book useful, though occasionally lacking in depth. (Jensen and Wirth's **User Manual and Report** is probably the next place to go.)

Well-designed examples demonstrate each topic clearly and reinforce concepts that have already been presented. Just as important, the examples avoid extraneous points. The illustrations and examples are nicely set off from the text.

The order in which topics are presented is ideal, especially for beginners: as much as possible, subjects are not used or referenced before being fully described.

Sets and pointers are usually the most difficult data structures to handle in textbooks. Zaks' approach is better than average.

Zaks describes sets well, but he fails to adequately describe why a programmer would choose sets over other data structures in a particular situation. Zaks also describes pointers well, using list structures as examples, but again fails to explain the broad range of uses for pointers. Some pointer examples are somewhat large.

The description of variant records is weak and does not include any examples of usage (just a simple example of declaration). Advanced examples are missing.

A few minor errors occur, some caused by recent shifting of the ISO draft standard. The answer section does not always cover enough of the exercises.

Overall, however, **Introduction to Pascal** is clearly organized, well written, and has complete coverage of the Pascal language. The book is excellent for beginning and intermediate programers.

(This book is also reviewed by Arthur Sale in the April 1981 — but just published — issue of the Pascal Users' Group Pascal News.)

— Will Neuhauser

Pascal-2 announced for RSTS, RT-11

Pascal–2 has now been released to RSTS/E and RT-11 customers.

The high-performance optimizing compiler is available on RSTS/E V6C and V7 and on RT-11 V3, V4, SJ, XM, and TSX-Plus. Pascal–2 was earlier released for RSX, RSX-11M, RSX-11M Plus, IAS, and VAX/VMS (compatibility mode).

Pascal–2 is a transportable five-pass compiler that emphasizes conformance to the Pascal standard while generating optimized object code. Written in Pascal, Pascal–2 will allow programs to be transported between computer systems with few changes.

The Pascal–2 compiler is larger and compiles more slowly than the Pascal–1 compiler, but produces code that is shorter and faster. Typical programs are 30 to 40 percent smaller and twice as fast. Error reporting is significantly improved over Pascal–1.

Discount Offered

Customers holding Pascal–1 licenses as of August 31, 1981, may claim a discount equal to 100 percent of the current Pascal–1 license price toward the purchase of an equivalent Pascal–2 license. For instance, the current price for a Pascal–1 U.S. site license is $1950. The same license for Pascal–2 is $3950. A Pascal–1 user can therefore purchase the Pascal–2 license for $2000. International customers will receive a similar discount toward their purchases, based upon international prices.

This discount will apply to all Pascal–2 orders received by December 31, 1981.

Design of Pascal–2 began in 1975. The first version of the compiler was completed in 1977. To include better ways of doing things, Pascal–2 has since been completely rewritten twice.

The Pascal–2 system consists of the compiler and the Debugger, in binary form; a number of utility programs, in source form; and the documentation, delivered in both printed and machine-readable form. The source versions of the compiler, Debugger, and run-time support library are available at extra cost, with a special license arrangements.

The Pascal–2 system has these major features:

- Code is optimized through constant folding, expression targeting, common tail merging, common subexpression elimination, and other techniques.
- Packed structures are implemented, saving space in the representation of data;
- A special "include" lexical directive allows the inclusion of multiple source files;
- A "structured constants" feature allows the initialization of data at compile time rather than at run time, thus saving space and making programming easier;
- Several low-level language extensions are implemented, including direct calls to assembly language or FORTRAN subroutines and direct memory addressing.
- Pascal–2 will give the source location for nearly 150 types of syntax errors and will detect many more run-time errors than Pascal–1.
- Strict interpretation of the emerging Pascal standard eases conversion of programs to other systems. Pascal–2 extensions can be flagged to indicate non-standard language usage.

The Pascal–2 system includes a high-level Debugger that helps programmers uncover errors that cannot be caught at compilation time. Pascal–2 also includes a collection of utility programs, written in Pascal.

The user manual is roughly twice the size of the Pascal–1 manual and contains many more examples.

Pascal–2, because it uses the same run-time support package as Pascal–1, is the logical next step for Pascal–1 users who want higher performance. Pascal–1 users, however, will continue to receive support and can look forward to further improvements to that product.

Newsletter No. 2 describes the Pascal–2 system in detail and lists the major differences between the implementation of Pascal–2 and of Pascal–1. These differences are also catalogued in the Conversion Guide, included in the Pascal–2 manual, which explains ways to convert Pascal–1 programs to Pascal–2 programs.

Note for RT-11 users

A single-density floppy (RX01) functioning as the system device is not large enough to run Pascal–2. An RX01 disk can be used as the release mechanism if the system device is double-density (RX02) or larger. Many users with RX01-based systems have found the step up to an RX02-based system well worth the modest cost. The only other RT-11 restriction is that the Pascal–2 text formatter, Prose, is too large to run under the XM monitor; Prose must run under the SJ monitor.

Letters

Editor's Note: Anyone who writes us expecting a technical reply can also expect the question to be published in this newsletter. (If you need an answer, others probably do too.) We will condense letters and correct grammar and spelling as needed. We will honor a request not to publish a letter, but we may generalize the question and run the letter anonymously.

To the editor:

We have been using the Pascal–1 compiler to teach Pascal for the last year, and I thought you might be interested in hearing our thoughts on the system.

First the details of our system and the compiler. The machine is a PDP-11/45 with 456K bytes of memory running under Version 7.0 of RSTS/E. We have been using Version 1.2E of the compiler. The compiler was used by 160 different students in two introductory Pascal courses.

Overall, we are reasonably happy with the performance of the compiler. It was reliable and performed efficiently considering the heavy work it was asked to undertake. We would like to continue to use the system in the future, but we are unlikely to move to Pascal–2. The reason for this is that "Pascal–2 is larger and compiles more slowly than Pascal–1" (Oregon Software Newsletter No. 2). Student programs tend to contain compilation errors, and even when they successfully compile they are unlikely to execute for very long. Thus the emphasis in Pascal–2 is the exact opposite of what we require in an educational establishment.

The following is a list of bugs or comments about the system:

(1) Would it be possible to extend the range of **set of char**? The lack of lower-case letters in sets is most regrettable.

(2) Run-time errors are reported with an indication of the program counter. From this there is no way of knowing where the error is located in the source. One way of overcoming this would be to have an extra column in the compilation listing containing the number of instructions generated up to the start of that line. The run-time system could also be made aware of the load point and by a simple subtraction could give the location of the error relative to the start of the program. The user could then discover from his compilation listing the line on which the error occurred.

(3) One of the most powerful features of Pascal is the subrange facility. This enables the development of reliable and readable programs. By not checking assignments to subrange variables either at compile time or run time the only argument for using subranges is for readability. Similarly no run-time checks are made on input to subrange variables. Checks on subrange variables are extremely important; this is the worst deficiency of your system.

(4) The correct response to a case expression being out of bounds is to give a run-time error, not to skip over the case.

(5) Integer overflow also needs to be checked. If you feel that all these run-time checks are expensive, they could be made an option, but by default switched on (the "putting to sea with lifejackets on" theory).

(6) The following program executes:

```
program pointers(output);
type
  arec = record i,j :integer end;
  aptr = ^arec;
var
  ptr :aptr;

  begin
    new(ptr); dispose(ptr);
    ptr^.i :=27
  end.
```

Why not check every pointer access to see whether it points inside the heap? If **dispose** changed the value of **ptr** to something outside the heap, then this sort of error could be detected.

Although the above might seem critical of your system, my main aim is to persuade you to develop your product from the current status of good to the status of excellent.

Dr. D. J. Robson
Computer Studies Group
University of Southampton
United Kingdom

Editor's Reply:

Perhaps our emphasis on performance of the code generated by Pascal–2 has left the impression that raw performance is the only new feature. Not so! Pascal–2 was designed

Continued on Page 6

with educational environments in mind, and with the knowledge that compilation speed, error checking, and run-time safety can be very important.

Thus, Pascal–2 checks for errors in the first two passes. If any are found, the compiler skips successive steps, including optimization. As a result, an incorrect program will compile nearly as fast with Pascal–2 as with Pascal–1.

More important is the problem of error checking in student programs. Pascal–2 detects many more syntactical errors and gives much more meaningful error messages than does Pascal–1. In addition, Pascal–2 detects a number of run-time errors that Pascal–1 does not. Pascal–2 is therefore much more likely to get beginners beyond syntactical blunders and common run-time errors and on to writing functional programs.

Other advantages of Pascal–2 over Pascal–1 are illustrated by your questions about limitations of Pascal–1:

Pascal–2 has 256-element sets, and Pascal–1 does not. Pascal–2 checks subrange assignments. Pascal–2 gives an error message when a case expression is out of bounds. Pascal–2 checks integer overflow for multiply and divide, which are the operations most likely to create an overflow.

Neither Pascal–1 nor Pascal–2 presently detects the pointer error on the PDP-11. However, upon inspecting the above example, we realized that Pascal–2 can be made to flag that particular misuse, at compile time. (It's an example of the undefined variable checking performed by the compiler.) We're working on that fix now.

The compile-time checking can find many, but not all, of these dynamic errors. Ideally, as you suggest, run-time checking should be available for all possible errors. However, the hardware design of the PDP-11 requires substantial overhead for some kinds of run-time error-checking, notably undefined variables and integer overflow. Other hardware designs, such as the VAX, permit more error checking at reasonable cost.

Your suggestions for changes in the listing to help detect run-time errors are valid. However, the interactive Debugger supplied with both Pascal–1 and Pascal–2 accomplishes the same thing. The Debugger automatically halts when it detects an error in the program and shows the source program statement at which the error occurs. Other Debugger facilities can then help the user determine the cause of the error.

John Anderson of the Zoology Dept. of the University of Washington in Seattle (USA) asks these questions after reading the Pascal–2 manual for RSX:

Q. I thought **forward** is standard, but Page 64 shows **forward** as a Pascal–2 extension. However, Page 70 shows no asterisk before **forward**, indicating that it is standard. Which is it?

A. **Forward** is a standard directive under the ISO standard. The problem is that the standard treats a directive as neither an identifier nor a reserved word. As described on Page 61 of the RSX Language Specification, we have chosen to treat **forward** as a reserved word. Therefore, our documentation is correct: **forward** is both a standard directive (Page 70) and a non-standard reserved word (Page 64).

Q. Also on Page 64, **emt** is listed as a predefined procedure, but it is not discussed anywhere. Shouldn't **emt** be explained in the section on low-level interfaces?

A. The **emt** ("emulator trap") procedure applies only to the RSTS/E operating system. The RSX and RT-11 manuals should have contained notes to this effect. The RSTS manual contains a brief definition of **emt** in the Language Specification; however, a more detailed **emt** section was inadvertently left out of the RSTS manual Programmer's Guide and is being sent to RSTS users as an addendum.

Q. The discussion (Page 55) of direct access does not make clear whether `/seek` is limited to 1 record/block. If `/seek` is limited to 1 record/block, what happens to long records?

A. The `/seek` switch refers to components in a file, not blocks on a disk. `/Seek` is **not** limited to 1 record/block, and records longer than a block are also supported. (In general, we've tried to document our Pascal products so that if a restriction is not mentioned, you may assume that unrestricted use is supported.)

Q. What happens with undetected run-time errors, such as `abs(-32768)`?

A. The most common cases involve integer overflow and uninitialized variables that escape detection by the compiler.

Signed integer overflow generally results in a value with a sign opposite from the mathematically expected result. For instance, the expression `abs(-32768)` yields -32768, not 32768, whereas the expression `maxint+2` yields -32767, rather than 32769. Unsigned integer overflow generally results in a value modulo 65536. The above is true for

addition and subtraction; results for multiply and divide depend upon which model PDP-11 processor is in use.

Uninitialized variables can cause unpredictable results, depending upon type. Pointers will often cause processor traps, since the value might be odd or might point outside the legal address space provided by the operating system. An uninitialized variable other than a pointer may cause unpredictable results, with the program "working" once and failing a second time, or failing differently each time it is executed.

The Xref utility can help uncover uninitialized variables. This cross-referencer marks each assignment to a variable; it is a good practice to check that each variable in a program is assigned a value at least once. This does not guarantee that the variable is initialized before use, but the approach can uncover some simple cases undetected at compile time.

'For' loop restriction

A **for** loop with an index variable that has been declared with an extended range will loop indefinitely upon occasion. The following example demonstrates the failing cases:

```
var i:0..65535; {an extended-range variable}

begin

 { Any final value less than 65535 works
    correctly, but the loop never terminates. }

  for i:=0 to 65535 do writeln('never halts');

 { Corresponding case for downto loops }

  for i:=10 downto 0 do writeln('never halts');

end.
```

The problem stems from the fact that the PDP-11 INC and DEC instructions do not set the carry condition code upon unsigned overflow.

This **for** loop restriction may be removed in a future release.

Information Exchange

If you need information on technical applications involving Pascal, or if you have an application that might be of interest to other users, send us a brief description for inclusion in the Information Exchange. Interested parties can contact one another directly. Follow the format of the items below.

Pascal Users Group for Pascal–1 and Pascal–2 is being formed to share the methodology in solving tasks with Pascal–1 and Pascal–2 and to spread information on problems and solutions with those products. Bruce Williams, Pascal User's Group, EOCOM, 15771 Redhill Avenue, Tustin, California 92680.

Portable database system (RSX, RT-11, Honeywell); 10 systems in the field. Dr. B. Gliss, c/o MPI Stuttgart, Heisenbergstr. 1, D-7000 Stuttgart 80, Germany. Tel. 0(711) 7830-251.

Q. We are using Pascal–1 (V1.2) on a PDP-11/23 with the RT-11 operating system and want to do interactive graphics. Is anybody using FORTRAN graphics subroutines in Pascal programs? Is anybody working on a Pascal graphics package? We also need device drivers for Tektronix equipment (4010, 4662). Gini Van Siclen (301-338-8344) or Dave Elliott (301-338-7049), Dept. of Earth and Planetary Science, Johns Hopkins University, Baltimore, MD 21218.

Menu-driven modeling and reporting systems designed for use by financial and planning oriented individuals; more than 200 routines in design and data manipulation, reporting and file handling; for PDP-11 (RSTS/E), VAX (VMS), Hewlett Packard 3000 (MPE). Canadian European Systems Ltd, P.O. Box 2884, Vancouver, B.C. Canada, V6B 3X4, (604) 732-9141.

Networking program that uses a master/slave distribution of processing; synchronous processing of messages received on multiple asynchronous ports; more than 90% in Pascal–1; EOCOM, a division of American Hoechst Corp., 15771 Redhill Avenue, Tustin, CA 92680.

Trouble with user trouble reports

A cautionary tale: On Friday, February 13, 1981, Joseph R. Smithson of Harnsford-on-Flushing in Middlesex, England, posted a trouble report about his Pascal–1 compiler to Oregon Software in Portland, Oregon, USA. In late March, Mr. Smithson began wondering about Oregon Software's claim of 30-day turnaround of trouble reports.

On Wednesday, April 1, Mr. Smithson sent a rather nasty wire to Oregon Sofware about the tardy response. And on Thursday, April 2, Mr. Smithson's original letter arrived at Oregon Software — via surface mail.

Yes, it really happens. We're less eager to admit that late responses are sometimes the result of our mistakes.

We want **all** Trouble Reports to receive a response within 30 days. Toward that goal we log every report and monitor its progress through our system. And even though our customer base has quadrupled in the past two years, we usually send an answer within 30 days. If you feel we've missed you, please call us collect or send us a Telex.

Steve Poulsen, a key member of our programming group, tells us that the main cause of delay in our response is the lack of sufficient documentation to permit us to reproduce the trouble. Will Neuhauser, who does preliminary screening and logging of each Trouble Report, tells us that slightly more than one-third of the reports are returned to the customer for additional information.

Overseas customers must use air mail or air express. In a rare case, a sample program may be short enough to Telex. But remember, our 30-day response guarantee starts when we receive the fully documented Trouble Report.

Other things you can do to help us serve you better include:

• Use our Trouble Report; send the reports to our sales department, which will coordinate our response. If you don't have a Trouble Report form, write anyway. Be sure to include your site number (e.g., #1-334); compiler version number (e.g., 2.0I); operating system version (e.g., RSX-11M-Plus V3.2), and a description that is sufficient for us to reproduce the bug or understand the problem. (And have us send you a pile of Trouble Reports.)

• Send the **shortest** sample program that shows the problem. Programs that are too long to enter onto our system by hand will be returned with a request that it be resubmitted in a machine-readable form.

A short example program demonstrating the problem can be evaluated much more quickly than a long program with "include" files and external functions and procedures. Interactive programs are especially difficult to analyze, so give sample input and be sure that the program provides prompts for all required input.

• If the program is more than a page, send it on an RT format floppy or a 9-track DOS format magtape. Place a written label on the media, with your site number, the format of the media (DOS, ASCII, FLX, RT-11), the density (single-density or double-density floppies, 800 or 1600 bpi tapes), the label used to identify the media to the system (if applicable) and the directory used to identify the media to the system. Don't send us confidential materials.

• If the bug is "fixed in next release", you can speed getting your free copy of the next version by having your Designated Contact Person sign the request on the Trouble Report before you send it.

Distributor customers

If you are a distributor customer, direct all Trouble Reports to your distributor instead of to Oregon Software. Our distributors have technical support staff to assist you, and our distributors will contact us in case of particularly prickly technical problems. Your distributor needs the same information that we do to process trouble reports: Site number, compiler version, operating system and version number, detailed descriptions in readable format.

Telephone requests

Before you call, ask yourself these two questions: Are you in support? Are you the Designated Contact Person? If the answer to both questions is yes, then call and ask for Technical Support. Be sure to have ready those magic numbers: site number and compiler version number (both at the top of any listing), and the operating system and version number.

We will try to determine the nature of the problem and give any quick fixes we know.

If, as a result of the call, you want an update to Pascal–1 or Pascal–2, the Designated Contact Person must request that update in writing. For legal reasons, we need a written request for each update, even though the update is free if you are in support.

If you aren't satisfied with your service, then call, write, or Telex directly to our president, Rusty Whitney.

Text formatting with Pascal-1

By DAVE THOMAS

(Mr. Thomas is with Novus Systems Technology Ltd., Koninginnegracht 56, 2514 AE, Den Haag, The Netherlands. In addition to the text-formatting system, Systems Technology has also used Pascal to develop a test and monitoring system for telex and telephone lines and to develop a telephone directory system.)

This article describes the use of Pascal for text formatting, specifically a set of programs collectively known as IGOR. This article may be of interest in two ways. First, some features provided in IGOR are normally found only on formatters running on large machines. Second, some implementation techniques described (in particular the support of variable-length strings in Pascal–1) may be of use to some readers.

Background

As a small software company, Systems Technology found itself spending a lot of time producing documentation. Typically in the course of a project we would write letters, tenders, contracts, various specifications, a user manual and a programmer's guide.

We looked for a text-formatting package available on our machines (LSI/11-23s with RX02 floppy diskettes or RL01 hard disks). However, we found none that met our primary requirements: document-level formatting commands; tailorability to our formats; support for simple graphics; support for various printers.

The first point was particularly important. The formatter was to be used by both technical and secretarial staff. Neither group would be willing to learn a complex series of commands to perform conceptually simple operations, such as starting a chapter or adding an element to a list of numbered points. As well as simplifying document preparation, this "high-level" approach ensures that all documents conform to a house style.

Given this apparent vacuum, we decided to implement our own formatter. (Besides, it was more fun.)

The IGOR Solution

We discarded the "word processing" approach ("what you see is what you get") because it entails losing too much information about the document structure. (Kernighan summarizes this loss as "what you see is all you've got").

Instead we decided to go for an off-line text formatter, much in the style of Script on the IBM 370 series or RUNOFF on DEC machines. An off-line text formatter processes the document, which contains formatting directives, to produce the final (printable) copy. The entire document is reprocessed after any change.

The IGOR system consists of a formatting program and a group of output processors, one for each device type. The formatting program takes a **source document** and a **document profile**, producing a formatted intermediate file. This intermediate file is run through the output processor appropriate to the device being used. The same intermediate file may be used many times, so that draft and final copy may be produced without reformatting the document.

Because most of our technical people enjoy a heritage of Script-like systems, we adopted the Script (and RUNOFF) philosophy for the source-document files. The document contains a mixture of text and formatting directives. An extract from a typical IGOR document is:

```
There are several objections to this
algorithm.
.sp;.ce ** more needed **
.Section 'Storage constraints'
On the target machine . . .
```

The text in the document is formatted according to the various format options. By default, text is word-wrapped between lines and justified to two margins. Commands are indicated by a special character at the start of a logical line. In the example above, the two commands **.sp** and **.ce ** more needed **** tell IGOR to space down one line and center the text (which is a note to the reader that part of the text is missing). The command **.Section** starts a new major section in the document with the given title. Text for the section then follows.

A feature possibly unique to IGOR is a simple means of specifying line graphics. Technical documentation often contains diagrams and tables using only horizontal and vertical lines, with the appropriate corners. IGOR allows these lines to be specified directly in the text of the document by means of a **graphics character**. The diagram is typed into the document via this graphics character wherever a line segment is needed. On output, IGOR looks at the neighbors of each graphics character to determine whether the character represents a line segment, an intersection or a corner. IGOR then does its best to represent the resulting graphic on the output device.

Continued on Page 10

As an example, a writer would type:

An **imbed** facility allows a document to be constructed from more than one source. Most of our manuals are split into chapters, one per computer file. This allows individual chapters to be formatted for proof-reading. Standard paragraphs in contracts and tenders may be imbedded in the same way.

Before the document body is processed, the IGOR formatter automatically reads the appropriate **document profile**. A separate profile is written for each document type (report, contract, etc.) The profile serves two main purposes. First, it establishes document characteristics such as page length and heading style. Second, it defines a set of document-specific commands. For example, the profile for reports will define commands for starting chapters and sections; the profile for letters will provide commands for generating greetings and farewells. New document types may be defined with appropriate profiles.

The intermediate file contains the formatted document in a device-independent form. The output processor for a particular device will translate this into a sequence of characters suitable for that device. One of the most important translations is the handling of line graphics. The serial printers we use for draft output do not support graphics. The output processor therefore translates the graphic sequences into available characters such as hyphens, vertical bars and plus signs. A rough version of document graphics may therefore be produced even on non-graphic devices.

The output processors are all screen-based. The document may be viewed, one page at a time, as it is being printed. Pages may be printed selectively or automatically.

Easily Extended

IGOR has features that make it easy to use and extend.

Variables: Arbitrary strings or numbers may be assigned to named variables. These may be manipulated and substituted into the document body. A common use of variables is in contracts. The name of the client is assigned to a variable at the start of the document. The standard paragraphs that follow may use this variable when they need to refer to the client and therefore need not be changed when a contract is drawn up for a different client.

Interaction: Text may be written to and read from the terminal during formatting. This allows data to be prompted for and included in the document, as well as providing a means of controlling the formatting itself.

Arithmetic: IGOR supports simple four-function integer arithmetic. As well as the more mundane uses, such as allowing chapters and sections to be numbered automatically, the arithmetic capabilities allow IGOR to be used as a simple calculator. Document profiles have been written to keep track of expenses and timesheets, with automatic calculation of purchase tax, subtotals and totals.

Conditional sections: Sections of the document may be conditionally excluded from the formatting process. One use of this is to produce two versions of the same document (one perhaps containing confidential material).

Additional commands: Formatting directives and text may be packaged together to form a **remote**. Remotes and IGOR commands are invoked identically, so that the user need not be aware of the distinction. When remotes are defined in the document profile, additional document-specific commands become available to the user. The `Chapter` command used in reports is actually a remote. It may be modified at each site to customize it to that site's requirements.

(Remotes may invoke other remotes, including themselves. This recursive property, combined with the conditional facility and variables, makes IGOR a usable programming language. There even exists a version of Towers of Hanoi written in IGOR.)

Remotes may be defined by writers of documents. Indeed, writers who are programmers often will expend more effort on the remotes than on the document itself. Occasionally a remote turns out to have general application, it which case we add it to a library. One such remote creates complex file description tables, automatically keeping track of field offsets within records. It probably took the programmer three or four days to get this remote right, but it has since saved us a lot of time and trouble. This facility also means that all the record descriptions in all our documents look the same, no matter who wrote them.

Implementation

The implementation of IGOR gave us the opportunity to learn a lot about using Pascal–1. We'll look at two aspects of the implementation in detail. The first is a "dirty trick" used right in the heart of IGOR to provide variable-length strings. Although the use is not something to be proud of, we feel that it may help other readers who have similar requirements.

The second aspect of the implementation is the management of the program source. We discuss both the compilation technique used and a method we used for verifying the correctness of the overlay structure chosen.

Continued on Page 11

Variable-Length Strings

Most of the data structures used in IGOR are lists or trees. However, it seemed more efficient to hold the values of variables and the bodies of remotes as arrays of characters. Because of space constraints, we needed to keep the storage used by each string to a minimum. The two normal solutions to this problem are linked lists of fixed-length arrays or allocation from a large static array. The first was too expensive in space. The second put an artificial limit on the amount of space available for variable storage. We therefore decided to implement true variable-length strings.

The solution we adopted is specific to Pascal–1, although the routines as developed could probably be linked into Pascal-2 programs with no ill effects. The general principle will work with any Pascal system that performs true run-time heap management.

A flexible string is defined to be a pointer to a record:

```
type
  FlexibleString = ^FlexibleStringRecord;

  FlexibleStringRecord
    = record
        Length  : FlexLength;
        Chars   : array [FlexLength] of char
    end;
```

where **FlexLength** is some suitably large subrange (we use 1..9999).

In the same way that normal dynamic heap items are created and deleted by **new** and **dispose**, we define two procedures **FNew(String,Length)** and **FDispose(String)** to handle flexible strings. The **FNew** procedure creates a flexible string record of the given length. The characters in the string are accessed as **String^.Chars[i]**.

Internally, **FNew** and **FDispose** use imbedded MACRO to access the Pascal run-time routines $B70 and $B72. $B70 is the routine called whenever a **new** is executed. It is passed the length required (rounded to a full word) and returns the address of a suitable free area on the heap. Both values are passed on the top of the stack. $B72 handles **dispose**. It is passed the size to free in register 0 and the address of the area to free on the top of stack.

To complicate things slightly further, we actually create the string 4 bytes longer than requested. Some of this extra length gets round a bug with the earlier versions of Pascal–1 that caused $B72 occasionally to free too much store. One additional byte is used to hold a sentinel. This is initialized by the **FNew** procedure. When **FDispose** is called, it checks for the presence of the sentinel value. If absent, it means that something attempted to write beyond the end of the string, and that program integrity has been lost.

The source of the two procedures is given on the accompanying page. It is probably unlikely that Oregon Software will support programs that use $B70 and $B72 in this way.

Managing the source

IGOR is implemented almost entirely in Pascal–1. The source (approximately 15,000 lines) is spread over about 80 RT-11 files. This makes the average file less than 200 lines long, resulting in easy editing and fast compilations. Extensive use is made of the external compilation feature of Pascal–1.

Each major procedure or function in IGOR is held in a separate source file and is compiled to produce a separate object file. The global constants, types and variables are held in three files. The PCL program supplied with Pascal–1 was modified to prefix these globals to the file being compiled. Thus to compile a component of the formatter, type:

```
.RUN IPCL <component name>
```

As space is a problem at run-time, the optimizer IMP is used when generating a production (as opposed to test) version of the IGOR formatter. The overall improvement in space was found to be about 6.8%.

The IGOR formatter is heavily overlaid (as the non-overlaid program size is about three times the size of the PDP-11's address space). The administration of a program with some 40 overlaid procedures, some recursive, is not easy! To ensure that the return path is never destroyed, we eventually resorted to drawing a 40-by-40 matrix of "what calls what". A quick program to find the transitive closure of this at least allows us to check that a particular overlay structure is valid.

(An aside to the interested reader with time to spare. A utility program that produces a list of procedures imported and exported by a Pascal program segment would be most welcome. The information needed seems to be in the symbol table file output by the compiler.)

Continued on Page 12

```
procedure FNew(var String : FlexibleString;
                   Size   : FlexLength);
var
  LocalString : FlexibleString;
  LocalSize   : PositiveInteger;

begin
  (* Round up size and allow for sentinel *)
  LocalSize := Size + ord (odd (Size)) + 4;

(*$C .GLOBL $B70
  MOV    LocalSize(6),%0      ; stack size to get
  MOV    %0,-(6)              ; on any m/c
  JSR    %7,$B70              ; call NEW
  MOV    (6)+,%0              ; pop address
  MOV    %0,LocalString(6)    ; and save it
*)
  (* Set length and sentinel *)

  with LocalString^ do begin
    Length := LocalSize;
    Chars[Length-2] := Sentinel;
    end;

  String := LocalString
end ;

procedure FDispose(var String : FlexibleString);

var
  LocalString  :  FlexibleString;
  LocalSize    :  PositiveInteger;

begin
  LocalString := String;

  if String = nil
    then <error action>
    else begin
      LocalSize := String^.Length;
      if String^.Chars[LocalSize-2] <> Sentinel
        then <error action>
        else begin
          (*$C
            .GLOBL $B72
            MOV    LocalSize(6),%0    ; size to free
            MOV    LocalString(6),%1  ; stack address
            MOV    %1,-(6)            ; on any m/c
            JSR    %7,$B72            ; call DISPOSE
          *)
          String := nil ;
          end;
      end;
end;
```

Conclusions

We were very pleased with the performance and reliability of Pascal–1 throughout the implementation of the IGOR system. The external compilation system, the ability to embed MACRO and the control possible with **reset** and **rewrite** were particularly appreciated.

However, lest success go to their heads, we do have several moans, none serious. We would have liked to use **set of char** several times, but were forced to code around it by the 64-element limit. Similarly, **packed** would save quite a bit of space at run time. Both of these features are included in the new Pascal-2 compiler.

The only other real complaint is user error handling. The run-time error procedure is passed an integer error code (good), but that code is always zero (not so good). It is useful to know what the error is before knowing whether the program can safely ignore it.

Despite these minor grouses, we are very happy with both Pascal–1 and the text formatter we implemented with it.

Editor's Note: This particular use of MACRO to access the Pascal run-time routines $B70 and $B72 is not unreasonable, but we caution any user about using embedded MACRO commands that depend upon our support library routines remaining constant forever. We cannot guarantee that will be true. (In particular, the internal calling sequences for `new` and `dispose` will be changed.)

Pascal–1 programs containing embedded assembly code can be linked to Pascal–2 programs with "no ill effect". See the Conversion Guide of the Pascal–2 User's Manual.

Systems Technology had to work around two problems that no longer exist. We've fixed the bug that required the variable string to be 4 bytes longer than needed. Also, beginning with V1.2H (and 2.0H), the error procedure code returns an integer value corresponding to the particular run-time error. Even so, the user's possible courses of action are still limited. See the Programmer's Guide of your User Manual for details.

As for Mr. Thomas's desire for a procedure cross-referencer, Pascal–2 has such an animal. As he notes, Pascal–2 also implements 256-element sets and packed structures.

The Log: Pascal-1 and Pascal-2

Oregon Software's first two Pascal Newsletters described the significant changes in Pascal-1 V1.2 from its initial release, V1.2A, in December 1979 through V1.2H in June 1981. This issue describes the significant changes in V1.2 through its present release, V1.2J.

This log also describes the changes in Pascal-2 since its initial release, V2.0H, in June 1981 through its present release, V2.0J.

To use this log, first determine what release of V1.2 or V2.0 you have. (The release number is printed on the headline of all program listings.) Then review the log to determine the changes made since the release of your version. If the changes are of particular importance to your application, the Designated Contact Person should request an update, in writing. The DCP is usually the senior programmer in charge of software. You will receive the latest version of the software.

The first section of this log describes changes applicable to Pascal-1 for all operating systems. Following sections describe Pascal-1 changes specific to each operating system. The version listed is the one in which the change first appears; succeeding versions also include the change. The same format follows for Pascal-2 (most Pascal-2 changes were for all systems, however.)

Future issues of the newsletter will continue to outline improvements to Oregon Software's Pascal products.

Pascal-1

V1.2J corrected these problems:

Incorrect code for some calculations

Compiler generated incorrect code for some mixed **real/integer** comparisons (operands were sometimes being reversed on the stack). Compiler also was not marking the result register of an **abs()** calculation as used, causing the register contents to be destroyed by later calculations.

Procedure parameter in nested scopes

Calling a procedure parameter from an inner scope didn't work.

Illegal scalar redefinition

Illegal redefinition of a scalar identifer as in **(red,white, blue)** as a procedure identifier (**procedure white**) would crash compiler with an odd address trap if the ordinal of the scalar was odd.

Changes to RSTS/E

None.

Changes to RT-11

V1.2I corrected these problems:

Unpredictable trap if file could not be opened

If the USR was swapped in when an error occurred, the default file channel blocks were not available. The **Error** procedure attempted to write to the default output file and produced a recursive error if the USR was in memory.

'Dispose of Nil' after reset/rewrite errors

In some cases when a program was unable to open a file, the additional error "Dispose of Nil" was generated while the program was being terminated. The error was caused by an attempt to release a buffer that had not been allocated.

'MMU Fault' on XM virtual jobs

If the first block allocated in the heap was later disposed of by an XM virtual job, a memory segmentation fault occurs, because a zone of unmapped memory exists between the program code and the heap. Linking with the /U:20000 switch will eliminate the problem in earlier versions.

Changes to RSX

V1.2J corrected these problems:

Extra 'readln' allowed

In some cases involving **text** files, an extra **readln** was being permitted at the end of the file without signaling a "Reading past end-of-file" error. The **eof** flag was properly being set, but the file pointer was not. An extra **readln** at **eof** now correctly gives an error.

Error with /APD switch

In some cases involving **text** files, the use of the append switch /APD would cause the support library to access an uninitialized variable. Users would see this problem as a "record length error". Depending on the uninitialized value being used, random memory may have been used as the line buffer, and this could cause traps and other random failures in the program. This problem happened because FCS rather than Pascal did the original positioning of the file.

Output errors with write

If more than 132 bytes were available in a file's block buffer, and if the last output operation was a **write** rather than a **writeln**, the end-of-file byte count in the file's header would not be properly updated; if the file were later opened, the data in the last line was not available to FCS. **Text** output files have been changed to always use move mode rather than locate mode.

Pascal-2

V2.0I fixed these problems:

Packed structures

Certain types of packed structures were not being unpacked properly; record packing was improved.

Structured constants

A structured constant containing variant records was not properly handled; a bug involving quoted string constants in constant structures was fixed.

Integer 'not' incorrect for constants

Folding problem with **not** operator was fixed. All constant **not** operations were being done as booleans ("not 0" became 1 instead of -1).

'In' operator erred

In operator incorrectly popped part of the set operand at times, leading to unpredictable fatal errors.

Integer 'and' gave no value

No value was generated for an integer **and** used in the context of an **if**, even when later code used the value as a common expression.

Function results passed incorrectly

A problem was fixed in the passing of a function result as a parameter. Problem occurred when two calls were nested and when the outer call returned a value shorter than the interior call (e.g., `writeln(trunc(realfunction))`).

Procedure parameters passed incorrectly

A problem was fixed in the passing of procedure parameters to inner procedures.

Functions as parameters

Error with function parameters resulted in functions erratically appearing to be **nonpascal** functions.

'For' statement code corrected

Three bugs were fixed in **for** statement code: **for** loops with a byte-sized control variable could give incorrect results; a **for** loop could terminate incorrectly; a **for** loop control variable was marked as undefined even if the **for** loop was contained in a conditional statement so that the **for** loop might not be executed.

Variable references erred

Intermediate variables were not being properly referenced after a non-local goto or a Pascal-1 call.

'Out of memory' with many real constants

An inconsistency in the handling of real constants was removed. The problem could cause "out of memory" errors in the compilation of programs using many real constants.

Reading of reals failed on non-FPP

Programs reading real numbers failed when compiled under /sim, /eis, or /fis, and a read of real numbers from a text file did not always store the result properly.

Large exponents caused traps

Large floating exponents could cause traps at compile time, and small numbers with many digits could be diagnosed as an exponent out of range.

Error reporting improved

A missing semicolon between parameter declarations is now detected; constant division by "0.0" now produces an error message; when an identifier is expected but not found in a type definition, user will correctly get "undefined identifier" message (previously, a "double definition" error would occur in unpredictable places).

Listings improved

Listings were corrected to properly handle comment nesting, tabs, and page ejects.

Reset/rewrite altered for RSTS/RT

`Reset/rewrite` calculations of file element length were changed to accommodate the way RSTS/E and RT-11 implement block files.

V2.0J fixed these problems:

Structured constant errors caused compiler loops

Error diagnostics in structured constants were not properly handled, causing the compiler to loop or crash. For instance, an undefined identifier in a structured constant would cause the compiler to loop.

Structured constants packed incorrectly

Structured constants were not being handled properly when an unpacked structure was included as a component of a packed structure or vice versa. This also caused Debugger problems. Code was added to make appropriate transfers when changing from one to another.

Duplicate definition caused compiler traps

Duplicate definition of an identifier caused an odd address trap.

Bad parameter specification caused compiler crash

If a user specified a parameter type by writing it out instead of using a type identifier, the compiler could crash before generating the proper error message.

Spooky errors with real constants

If a constant happened to have the same binary value as an instruction that could be deleted, the constant could be deleted as an "optimization", producing strange errors.

Bad code for integer expressions

Incorrect code was generated for some integer expressions that used both multiplication and division.

'In' operator error

If the left-hand operand was constant and right-hand operand was both stored in a register and 9 to 16 bits long, then the generated code failed.

Errors with set expressions

Certain complicated set expressions would cause errors.

More 'for' loop problems

Loop variables were inaccessible from external procedures, and some `for` loops failed to terminate properly.

Problem with 'nonpascal' real functions

A `real` result from a `nonpascal` function was mistakenly assumed to be returned in the floating-point accumulators. All `nonpascal` function results are now taken from the general registers.

External functions

An external function was not allowed as an actual parameter.

Loophole produced errors

`Loophole` produced erratic results if the parameter to `loophole` was shorter than the result (e.g., when an integer was floated).

Trunc and round

`Trunc` and `round` did not work on non-FPP machines.

'Undeleted temps' message

Internal compiler errors generate this message. In some cases, the compiler failed to delete temporary variables (usually generated during optimizations); in others, the compiler lost track of the stack.

'Out of memory in procedure x'

A number of internal changes were made to allow the compiler to handle more complex user programs.

Error reporting improved

A colon after **otherwise** (a common mistake) generates a specific error message rather than a series of general error messages.

Unfixed

Incorrect error for overly complex programs

Deeply nested records will give the message "Stack overflow. Try expanding $$heap xxxxxx". The message should say simply "out of memory". The problem is a limitation on total memory storage; the solution is to reduce the number of types or the number of nesting levels.

UIC in default fails in reset/rewrite on RSX

The Pascal program cannot locate the file when a UIC is specified in the third parameter of **reset/rewrite** (the default file name). The problem is with the way FCS PARSE handles defaults. Temporary solution: don't specify UICs in the default name parameter.

TSX and TSX-Plus

A program may run out of memory under TSX unless the user runs the TSX-supplied SETSIZ program. This program sets the size of memory required by the program into byte 56 of the .SAV file.

The installation process links the PCL program with a /STACK specification, which is not allowed under TSX. Simply remove the /STACK switch and relink PCL.

RT-11 disk space problems

Occasionally, an RT-11 program that has worked consistently in the past will produce the error messages "attempting to read past end of file" or "not enough room for user on device", or will fail attempting a **rewrite** file operation. A related problem is the Pascal–2 compiler error message "rewrite failure: CACHE.TMP".

The problem may be caused by a lack of contiguous disk space for a file. The significant term here is "contiguous": be several thousand disk blocks may be available, with no single space that is large enough for the file.

The cure is to squeeze the disk with the SQUEEZE command. This command moves all of the existing files to the head of the disk, compressing the unused blocks into one contiguous region on the disk. The command is "SQUEEZE XXX:", where XXX: is the disk to be compressed. (The system will ask you to confirm the SQUEEZE command.)

Warning: Never squeeze a disk under TSX while other persons are using the disk. You can destroy their files.

A further consideration of disk space: Unlike RSTS and RSX, RT-11 files cannot be extended after they are created. An application that will accumulate data over several runs should, when first executed, create files that are large enough to contain all of the expected data.

Linker transfer address problem ... again

The latest revision (V6.01E) of Digital's Linker still has a problem setting the transfer address of Pascal programs. (The problem is with overlaid programs that have their transfer address defined from a library.) The V3 Linker does not have this problem and may be used under V4; users who have only the V4 Linker should explicitly set the transfer address as follows:

```
.R LINK
*PROGRAM = PROGRAM,SY:PASCAL/T
Transfer Address ? $START
^C
```

Pascal-2 too big for SJ BATCH

The RT-11 SJ monitor does not leave enough memory for a user to submit Pascal–2 compilations as BATCH jobs. (We're working on it.)

Errors, additions to manuals

Pascal-2 RSX manual

I/O Control Switches, Page 20

Add this switch:

/buff:n (Buffersize): Pascal–2 normally allocates the minimum space required for a file buffer, which is usually 512 bytes but is dependent on device and file characteristics. More efficient I/O transfers can be performed at the cost of additional memory. The /buff:n switch specifies the storage (in decimal bytes) to be allocated to a file buffer.

Overlays, Page 23

The correct line in the ODL example should be:

```
.ROOT SINGLE,SYSIO,*MAIN-*(SUB1,SUB2,SUB3)
```

The Stack, Page 25

Second paragraph, first sentence should say: "The space allocated for the stack ... ", not "the stack frame".

Storage Allocation, Page 27

Second paragraph, first sentence under Set Types should say: "In this example, **Hotset** is allocated the same amount of space as **Colorset**", not "the same space".

Structured constants, Page 54

The structured constants example in the Language Specification is incorrect. Parentheses are needed around each record in the outer array. The following is the correct text of the example.

```
type
  compensation = (paid, unpaid);

  paytype = record
    title : (clerk,indian,chief,president);
    case compensation of
      paid : ( rate : real );
      unpaid: ();
    end;

  employeetable = array[1..4] of record
     name : packed array[1..10] of char;
     payinfo : paytype;
     end;

const
  workers = employeetable(
    ('Charlie   ',(clerk,Paid, 3.40)),
    ('Samuel    ',(indian,paid,5.25)),
    ('Maxine    ',(chief,paid, 6.85)),
    ('Edward    ',(president, unpaid))
    );
```

Reset of an undefined file, Page 61

Reset of a nonexistent file generates an error message. In versions of Pascal-2 previous to 2.0J, the **reset** would create an empty file.

Emt procedure, Page 64

Predefined **emt** procedure is valid only for RSTS/E systems; the RSX manual should have contained a note to this effect.

Installation Guide, Page 173

PROCRF: should be PROCRE:

Installation Guide, Page 175

The Installation Control File (Appendix A) contains the line:

```
;       DEBUG.OLB
```

This line should be:

```
;       DEBUG.ODL
```

Pascal-2 RT-11 manual

Emt procedure, Page 60

The predefined **emt** procedure is valid only for RSTS/E systems; the RT-11 manual should have contained a note to this effect.

Prose Guide, Page 132

The guide should note the restriction that Prose is too large to run under the XM monitor.

Installation Guide, Page 170

The command:

```
ASSIGN  SY:  DK:
```

should be:

```
ASSIGN   SY  DK
```

RT-11 V4 users can use the COPY command in place of R PIP to copy the Pascal files, e.g.:

```
.COPY
From? MTO:*.*
To?   SY:*.*
.COPY LIBFPP.OBJ PASCAL.OBJ
.COPY SJ.SAV PASCAL.SAV
```

Similarly, on V4, the DELETE command can be used to clean up the system disk after installing the Pascal system. The format is:

```
.DELETE SJ.SAV,XM.SAV
```

See the Sample Installation Commands for details.

Pascal-2 RSTS/E manual

System error procedure, Page 33

The section should end with this note:

The error procedure forms part of the Pascal run-time system that is shared by all RSTS/E users and is not normally replaced on a system-wide basis. Programs requiring the replacement of the error procedure must be compiled as separate run-time systems, by means of the /rts switch.

Emt procedure, Page 57

The predefined **emt** procedure is defined in the Language Specification, but a longer section with several examples was left out of the Programmer's Guide. This section is being sent to RSTS/E users.

All Pascal-1 manuals

Identifiers and MACRO instructions

If a program contains embedded assembly code, no identifier in that program can have the same name as a MACRO instruction. For example, a constant named **mov** will conflict with the embedded code mov R0,-(sp). In addition, a function or procedure that has the same name as a MACRO instruction and is declared as external will generate an error when assembled.

For example, this program creates a MACRO-11 error:

```
function sp:boolean;external;

begin
  if sp then writeln('It works');
end.
```

Exiting from the Debugger

The **Q** command or a Control-Z (^Z) will exit from the Debugger.

Pascal-1 RSX manual

I/O Control Switches, Programmer's Guide

Add this switch:

/buff:n (Buffersize): Pascal-2 normally allocates the minimum space required for a file buffer, which is usually 512 bytes but is dependent on device and file characteristics. More efficient I/O transfers can be performed at the cost of additional memory. The /Buff:n switch specifies the storage (in decimal bytes) to be allocated to a file buffer.

Overlays, Programmer's Guide

The correct line in the ODL example should be:

```
.ROOT SINGLE,SYSIO,*MAIN-*(SUB1,SUB2,SUB3)
```

Pascal-1 RT-11 manual

Installation Guide

The Pascal–1 installation program for RT-11 requires a machine with at least 24KW (48KB) of user memory. This causes problems on small RT-11 SJ systems and on all TSX systems. The problem occurs while the installation program copies files from the distribution medium to the system disk. Circumvent the problem by copying the required files manually. This restriction will be fixed in the V1.2K release of Pascal–1.

Oregon Software

Printed in USA

Pascal
NEWSLETTER

NUMBER 4 OREGON SOFTWARE AUGUST 1982

Pascal-2 Cross-compiler Released

Oregon Software is now releasing a Pascal–2 system development package to major RSX users seeking a lead position in the Motorola MC68000 market. The package consists of an RSX-based cross-compiler that generates code for the MC68000 and a "stand-alone" concurrent programming package for the MC68000.

The package will not be available to end users until the third or fourth quarter. Oregon Software is making the preliminary version available for OEM customers because of the significant lead time required to bring out major products on the new processor, which does not yet have a good development environment.

Oregon Software's goal is to give OEMs a fast and dependable tool with which to develop the first wave of products for the MC68000; to encourage OEMs to distribute Pascal–2 as part of their final packages; and to help OEMs begin developing other cross-compilers having the MC68000 as a target.

The RSX-based Pascal–2 cross-compiler will run on standard RSX systems, including the VAX in compatibility mode. The cross-compiler generates object files that can be linked and run under the VERSAdos operating system on the MC68000. The package includes the support library and XFR, a utility that can transfer object files or text files between RSX and VERSAdos.

The MC68000 stand-alone package allows concurrent programming in standard Pascal, using monitors and predefined "primitives" to synchronize concurrent processes. The package allows true priority scheduling and the ability to write device drivers in Pascal. Sources are included. A PDP-11 stand-alone system is scheduled for release later this year.

Also available for OEMs now is a VERSAdos native Pascal–2 compiler, processing the same language as the cross-compiler. The package includes the support library, the Debugger, Profiler, and the rest of the standard Pascal–2 utility package, running on VERSAdos.

The RSX cross-compiler and the VERSAdos native package are sold separately. The stand-alone system, which consists of library routines that enable the compiled programs to run without an operating system, can be purchased only as an additional option to one of the compilers.

Release of the end-user product depends upon Oregon Software's development or acquisition of a cross-linker and cross-assembler for the MC68000. Until that time, a user must work with the cross-compiler on both the RSX system and an EXORmacs or use the native MC68000 compiler on an EXORmacs alone.

Prices for the RSX cross-compiler and VERSAdos compiler are comparable to Oregon Software's current prices for PDP-11 products. Purchasers of the OEM package will receive a 100 percent credit to upgrade to the standard release product when it is available for end users. OEMs are eligible for volume discounts.

OEMs should contact Paul deBruyn, Oregon Software's OEM marketing manager, for technical details and for more information on prices and delivery.

The cross-compiler is available on 9-track magtape, 800 and 1600 bpi; floppy disk, single and double density; and RK05, RL02, and RL01 cartridge disks. The native compiler is initially available only on floppy disk.

Documentation includes a VERSAdos user manual that is similar to the PDP-11 manuals for Pascal–2, but includes sections on cross-compiling and concurrent programming.

Product Enhancements

The cross-compiler and MC68000 native compiler feature an error "walkback" feature that automatically prints the

Continued on Page 2

In this issue...

RSX cross-compiler released . . . . . Page 1
Information exchange . . . . . Page 2
Random number generator . . . . . Page 3
Faulty shipments . . . . . Page 6
ROM applications . . . . . Page 7
Distributors' seminar . . . . . Page 14
RSX system commands . . . . . Page 15
RT-11 file variable . . . . . Page 18
The Log: Pascal-1 and Pascal-2 . . . . . Page 19
RT-11Quirks . . . . . Page 21
Pascal calls FORTRAN . . . . . Page 22
Additions, corrections to manuals . . . . . Page 22
Letters . . . . . Page 23

execution history of the procedures that lead to a run-time error. This capability will be added to the Pascal–2 products for the PDP-11 in the near future.

The cross-compiler also includes XFR, a data transfer utility, that transfers object files or text files from the RSX system to the VERSAdos system or vice versa. XFR uses a "hand-shaking" format to guarantee accurate receipt of data and can produce a log on RSX of program execution and other steps on VERSAdos.

Language

Except for the `reset/rewrite` commands, which accommodate differing file system accessing options, the language for all the Pascal–2 compilers is host-independent and virtually identical for all systems. The compilers support all capabilities of standard Pascal and conform closely to the current draft of the proposed Pascal standard (International Standards Organization dp7185).

The front end is nearly the same for each processor/operating system combination. The command-line interface is different, to correspond to the operating system on the host. The support library is different for each operating system. The code generator is different for each processor or for an operating system with a different object format or assembler than that of the manufacturer.

OEM Manager Hired

Oregon Software has hired an OEM marketing manager, Paul deBruyn. Reporting directly to company president Rusty Whitney, deBruyn is responsible for contract negotiations with large OEMs, recruitment of new accounts, and technical support of existing accounts.

Before joining Oregon Software, Paul deBruyn directed contract administration and managed hardware research and development with Automatic Data Processing (ADP) in Portland. In 1977, he was Portland branch manager for the Computer Systems Division of Memorex Corp., involved with direct end-user sales of IBM plug-compatible peripherals. His background also includes several years as a data-processing consultant for Sperry Univac and as a systems specialist for RCA Computer Systems Division.

Oregon Software established the new position to broaden its OEM customer base. Most of the company's 3,000 customers are end users.

In conjunction with developing new markets, Oregon Software also has commissioned a marketing survey and is seeking a director of marketing and sales with a strong software background.

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might be of interest to other users, send us a brief description for inclusion in the Information Exchange. Your description should follow the format of the items below. Interested parties can contact one another directly.

Quick Task Builder (QTB) for RSTS/E V7.0 replaces DEC's Task Builder, contains all the same functions with additions such as clustering run-time systems, runs three or four times faster. Commercial Computer Services Inc., 3000 Dundee Road Suite 402, Northbrook, IL 60062.

Pascal Record Management-11 for RSX-11M and RSTS/E; source written in Pascal-1; installation on RSTS cannot use command files for library build; documentation on magnetic media; available through DECUS Program Library Software, Catalog number: 11-479 PRM-11. Kenneth G. Tibesar, 3M Company, BCP Lab, Bldg 235-2F, St. Paul, MN 55144.

RMS/Pascal Interface for RSTS/E; allows storage of large amounts of data with easy access thorugh RMS indexed files for scientific/engineering or commercial data processing applications; 25 interface routines compatible with Pascal-1 or Pascal-2 (V2.0J or later for RSX); runs on PDP-11 LSI-11 through 11/70; uses EIS or FPP instruction set. Valley Software Inc., 390-6400 Roberts Street, Burnaby, B.C. Canada V5G 4G2, (604) 291-0651.

Interface for RMS-11K and Pascal-1 Version 1.2; major file system has 4 indexes; uses FORTRAN callable routines from DECUS (No. 384) written in MACRO-11; high-level interface based on Basic-Plus-II statements for RMS operations. Mr. H. D. Rees, Taylor-Minister Acugreen Lsg. Ltd., Bush House, Room 2, Merwood Avenue, Oxford, England.

Pascal Users' Group for Pascal–1 and Pascal–2 has been formed. Interested parties and prospective members may contact: Bruce Williams, Pascal Users' Group, EOCOM, 15771 Redhill Avenue, Tustin CA 92680.

Robert S. Suddath of Harmon Electronics sends information for RSX-11 users. The switch `'\ACTL:101000'` allows several tasks to access a file and not leave it locked if one of them aborts. This switch sets bits 9 and 15 of F.ACTL in the File Descriptor Block (FDB) as described in the DEC I/O Operations Reference Manual, Appendix A.

Random Number Generator for Pascal

By D. John Anderson

(Mr. Anderson is with Solaster Software Corporation in Seattle Washington, where he writes applications programs for microcomputers. He developed this random number generator for the Zoology Department at the Unversity of Washington, where he used the computer for his research in theoretical biology.)

```
Anyone who considers arithmetical
methods of producing random digits
is, of course, in a state of sin.
-- John Von Neumann (1951)
```

Although Pascal does not support a predefined random number generator, a good source of random numbers can be of critical importance. In creating one, it is important that you understand the limitations of the random number generator you use. This article will show how to produce good (but not perfect) random numbers for programs compiled with Oregon Software's Pascal–1 on PDP-11 computers.

No algorithm produces perfect random numbers. In *Seminumerical Algorithms*, however, Donald Knuth discusses the advantages and limitations of most widely known algorithms. Of these, the linear congruential method provides the "nicest" and "simplest" random number generator for the machine language of most computers.[1] The linear congruential sequence is probably the best understood and most commonly used algorithm for random number generators. Its properties are well-known, and it can be programmed easily in the assembly language of most machines, making it significantly faster, which may be an important factor in some compute-bound simulations. But, the linear congruential method also has some disadvantages. The integer arithmetic requires at least 30 bits of precision, which is often more difficult to do in a portable high-level programming language than in assembler.

In summary, the linear congruential method follows this scheme. Some initial value of the integer X is chosen to be the "seed". The next random integer is calculated as

```
X := (A*X + C) MOD M
```

where A, C and M are constants having special properties (described further on). When a random number is required, this formula is applied to the last random number to produce a new one.

You must follow several guidelines when using the linear congruential random number generation method.

1. The seed, or initial value of X, may be any integer. To repeat the same sequence of random numbers, you can start with the same value for the seed. To choose a different sequence each time your program runs, you can set the seed to the time of day.

2. The value of M should be at least 2 raised to the 30th power or larger. This presents a problem since Oregon Software Pascal uses 16-bit integers. The obvious solution is to implement 32-bit integer arithmetic using two 16-bit integers. This can be done in either Pascal or assembler. I chose assembler because execution speed was critical to our application.

3. The multiplier, A, is probably the most difficult choice. Choosing `A MOD 8 = 5`, as long as C has no common factors with M, ensures that the random number generator will produce M different values before it repeats. Also A should be between .01 and .99 of the value of M and contain no regular patterns of binary or decimal digits.

   Of all the test procedures for verifying a random number generator, the spectral test is perhaps the most difficult to pass. Knuth discusses this test and uses it to examine various multipliers. Two multipliers that he discusses pass the spectral test and meet all the criteria we have established so far: 1,812,433,253 and 1,566,083,941 (base 10).[2] Arbitrarily, I chose the latter for the random number generator presented here.

4. The value of C should have no common factor with M, so I chose `C=1`.

5. For most applications, a random real number between 0 and 1 is preferable to a random 32-bit integer. Since the most significant bits are more random than the least significant bits, the 32-bit integer should be considered a fraction between 0 and 1 with the binary point in front of the most significant bit.

Even after you have carefully made these decisions, the random numbers may not be good enough for some rare applications. If you think your application may be dependent upon the quality of random numbers, try two different methods (or multipliers) and compare the results. According to Knuth "every random number generator will fail in at least one application".[3]

Program UNIF is an implementation of my random number generator for Pascal–1. The comments in the program further explain its calculations. UNIF can easily be adapted for use with Pascal–2 on the PDP-11. As written, the uniform random number generator uses the extended instruction set (EIS) and is intended to be compiled as an ex-

Continued on Page 4

ternal procedure. This program may be added to a library of external functions and procedures.

Making sure that this program actually produces good random numbers is not simple. The problem has two parts: First, the algorithm must really be a good random number generator; second, the code must correctly implement the algorithm.

I have tested this program on both criteria. For guidance to a correct algorithm, I have primarily relied on the authority of Donald Knuth. In addition to the spectral test that he calculates for this generator, I have done autocorrelation tests and chi square tests for uniformity. This generator, and others, have been used for the past two years in numerous simulations, and we have detected no problems. If one looks hard enough, however, it will always be possible to show that any pseudo-random number generator is non-random. To verify the code, the program was independently implemented by two different programmers on two different machines (an LSI-11 and a Z80) and the first million random numbers were checked to make sure they agree. In addition, selected random numbers were calculated by hand and compared to those calculated by the program.

NOTES

1. Donald Knuth. *Seminumerical Algorithms*, volume 2, second edition, in his series: *The Art of Computer Programming*, (Reading: Addison-Wesley, 1981), page 155. Most of the information in this article is based on pages 1-170 of this book.
2. *Seminumerical Algorithms*, pages 89-113.
3. *Seminumerical Algorithms*, page 156.


```
program Unif {Uniform random number generator} ;

  type
    LongInteger =
      record
        Low, High: Integer
      end;

    {$E+ Treat this as an external procedure}


  function Unif(VAR X {the seed} : LongInteger): Real;
  {
   * Linear Congruential Random Number Generator:
   *
   *          X  :=  (A*X + C) MOD M
   *
   *    where:
   *
   *        M =          2**32 (base 10)
   *          =  4 294 967 295 (base 10)
   *          = 40 000 000 000 (base 8)
   *
   *        A =  1 566 083 941 (base 10)
   *          = 13 526 105 545 (base 8)
   *
   *        C = 1
   }

    var
      Temp: Real {Used to pass the result to the caller} ;

    begin {Unif}
```

```
{$C                                ;Insert assembler code
;
        LowA  = _^O 105545         ;Low  16 bits of A in octal
        HighA = _^O 056530         ;High 16 bits of A in octal
;
; R0-R1 will be used as a 32-bit accumulator to carry
; out the A*X multiplication as follows:
;
;                          High A   Low A
;                        x High X   Low X
;                        ----------------
;                          Low A  * Low X
;                 High A * Low X
;                 High X * Low A
;    + High A * High X
;   ------------------------------------
;                 64-bit product of A*X
;                                   + C
;   ------------------------------------
;           64-bit product of (A*X) + C

; Since we need to calculate this product modulo 2**32 (i.e. throw
; away the 32 most significant bits) then the low 32 bits of the
; 64-bit product A*C is simply:
;
;          Low A * Low X
;       + (Low order 16 bits of (High A * Low X) ) * (2**16)
;       + (Low order 16 bits of (High X * Low A) ) * (2**16)
;
; We don't even need to calculate High A * High X.  In practice,
; adding C (which in this case is 1) can be done before or after the
; modulo operation, since only the lower 32 bits of any arithmetic
; operation are retained by the program.
;
        MOV     X(SP),R4           ;R4 := address of X (the seed)
        MOV     (R4)+,R3           ;R3 := Low X
                                   ;R0-R1 is used as a 32 bit accumulator.
        MOV     R3,R0              ;R0 := Low X
        MOV     #LowA,R2           ;R2 := Low A
        MUL     R2,R0              ;R0-R1 := Low X * Low A
        ADD     R3,R0              ;Convert the result of the
        TST     R3                 ;  signed multiply into the
        BGT     Skip               ;  result of an unsigned
        ADD     R2,R0              ;  multiply.
Skip:   MUL     #HighA,R3          ;R3 := High A * Low X
        ADD     R3,R0              ;Add the low 16 bits of (High A * LowX) to the
                                   ;  high 16 bits of the R0-R1 accumulator.
        MOV     (R4),R3            ;R3 := High X
        MUL     R2,R3              ;R3 := High X * Low A
        ADD     R3,R0              ;Add the low 16 bits of (HighX * Low A) to the
                                   ;  high 16 bits of the R0-R1 accumulator.
        INC     R1                 ;Add C to the R0-R1
        ADC     R0                 ;  accumulator.
        MOV     R0,(R4)            ;Replace the old seed
        MOV     R1,-(R4)           ;  with the new seed.
```

Continued on Page 6

```
;Convert the high order bits of the 32-bit integer to a
;single-precision floating-point number between 0 and 1.
        MOV     #^O200,%2       ;Set exponent to 1.
        SEC                     ;Set the carry bit so shift
                                ;  will terminate if seed is 0.
Loop:   ROL     R1              ;Shift left one bit at a
        ROL     R0              ;  time until normalized.
        BCS     EndShf
        DEC     R2              ;Decrement exponent for
        BR      Loop            ;  each shift.

EndShf: CLRB    R1              ;Move low order byte from R0
        BISB    R0,R1           ;  to R1 without sign extend.
        SWAB    R1              ;Store low order 16 bits of
                                ;  32-bit floating point number (shifted one bit left).
        CLRB    R0              ;Store high order 16 bits of
        BISB    R2,R0           ;  32-bit floating point number
        SWAB    R0              ;  (shifted one bit left).
        ROR     R0              ;Shift one bit right.
        ROR     R1
        MOV     R0,Temp(SP)     ;Store the result (random number
        MOV     R1,Temp+2(SP)   ;  between 0 and 1) into TEMP.
  }
      Unif := Temp {Return the result of the function}
    end {Unif} ;
```

Faulty Shipments

We know how frustrating it is to receive software and then be unable to read the release media. It happens to us, too. Sometimes the problem is "bad media", whether it be hard disk (RL01/RL02 and RK05), floppy disk (RX01), or magnetic tape. Other times, however, the problem is one of compatibility between your system and ours, and the solution can be elusive.

Technically, the term "bad media" means the magnetic coating on a disk is too thin to permanently store information. Although a faulty shipment may occur because of bad media, such problems are relatively rare.

The causes. A faulty shipment is often caused by hardware errors, either on your end or ours, or both. Recent problems arose because the read/write controller board failed intermittently on Oregon Software's No. 1 floppy disk drive, trashing odd-numbered floppies for several customers. At times, a customer needs to align the heads on his drive so the release media can be properly read. Occasionally, we receive calls about "bad media" because the customer does not set up his system to read the format we supply. For example, customers may not realize that they cannot boot our floppy disks, and that they must use the FLX conversion from DOS format for RSX releases (see your User Manual for details). In some cases, the media is "zapped" in transit. Also, the temperature of the media should be the same as the computer room, so immediately mounting a disk or tape that's just come from the cold outdoors could give faulty results.

What to do. Bad shipments should be reported to Oregon Software immediately. When you call, be prepared to answer the following questions, which will help us determine a pattern and (we hope) diagnose the problem:

- What was the exact error message and the name of the file?
- If you received the compiler on floppy disk, what is the number of each floppy disk that cannot be read? For example, number 3, number 6.
- Can you read Digital's release media?
- Who manufactures the disk drive used to read the media?
- Who maintains your equipment?
- How often is maintenance performed?

Your call will be taken by a programmer, who needs your answers to these questions to help root out the problem. If the cause appears to be at our end, we will immediately reship the order. We ask that you return the faulty media with a letter answering the above questions and stating that no copies of the software have been retained. We will reship the entire package on the same medium, or another if so requested. If the problem conclusively lies at your end, we will do all that we can to help you solve it.

Pascal-1 for ROM Applications

by J.E. SANDER

Editor's note: This article is reprinted from the IEEE Transactions on Nuclear Science, Vol. NS-29, No.1, February 1982. This work was supported by the Department of Energy under Contract No. DE-AC03-76NV0-1183.

Introduction

This paper describes a technique for developing code in the Pascal-1 programming language which will execute successfully in read-only memory (ROM). Along with the capability of executing Pascal code in ROM, the technique provides one with the advantage of being able to write stand-alone, microcomputer-based software in a high-level language.

The information in this paper is based on the author's experience with a microcomputer-based control and monitor system developed for the Nuclear Test Program at the Lawrence Livermore National Laboratory (LLNL). This system executes Pascal-1 code residing in erasable programmable read-only memory (EPROM), and it works!

The software is written in Pascal-1 V1.2C, and is compiled and linked under the DEC RT-11 operating system. This software, a simple operating system in itself, executes in a stand-alone configuration. The system hardware consists of a DEC LSI-11/23 processor, 16K of ROM, 16K of core memory, and a DEC VT100 terminal.

Developing Code to Reside in ROM

There are four items one must consider when developing software that will reside in ROM and execute in a stand-alone configuration. First, the system run-time memory organization must be known. Second, all read/write data must be located outside of the ROM area. Third, all I/O functions must be coded without using the features of an operating system. In addition, program instructions that issue operating system calls cannot be used. Fourth, the coding of interrupt service routines and vector initialization require certain precautions.

Run-Time Memory Organization

The Pascal-1 system discussed here runs under the DEC RT-11 operating system. Its run-time memory organization is defined according to Figure 1-A. The RT-11 vector partition, starting at location 0, contains interrupt vectors and a system communication area used by the RT-11 monitor. The next partition contains the user program code and is variable in size. The Pascal system variable, `$BEGIN`, defines the starting address of the user's main program block. Following the program code is the Pascal-1 system partition. The Pascal-1 Support Library routines, contained in the file PASCAL.OBJ, are located here. The next partition is for dynamic memory allocation and is variable in size. Register 5 (`R5`) points to the bottom of this partition and is defined as the global variable base. All global variables are stored in this area and are accessed through `R5`. Global variables are those defined in the main program block. The stack pointer (`SP`) is initially set to the top of this partition. Upon entry to each block, a stack frame is pushed on the stack and each frame is popped off the stack on exit from the block. A stack frame contains temporary storage for calculations, parameter passing, block linkage, and local variables. Local variables are those defined in each block, except for the main block. The last two partitions consist of the RT-11 monitor and the I/O page. For a more detailed description of these partitions, refer to the Pascal-1 Version 1.2 RT-11 manual.

Read/Write Data

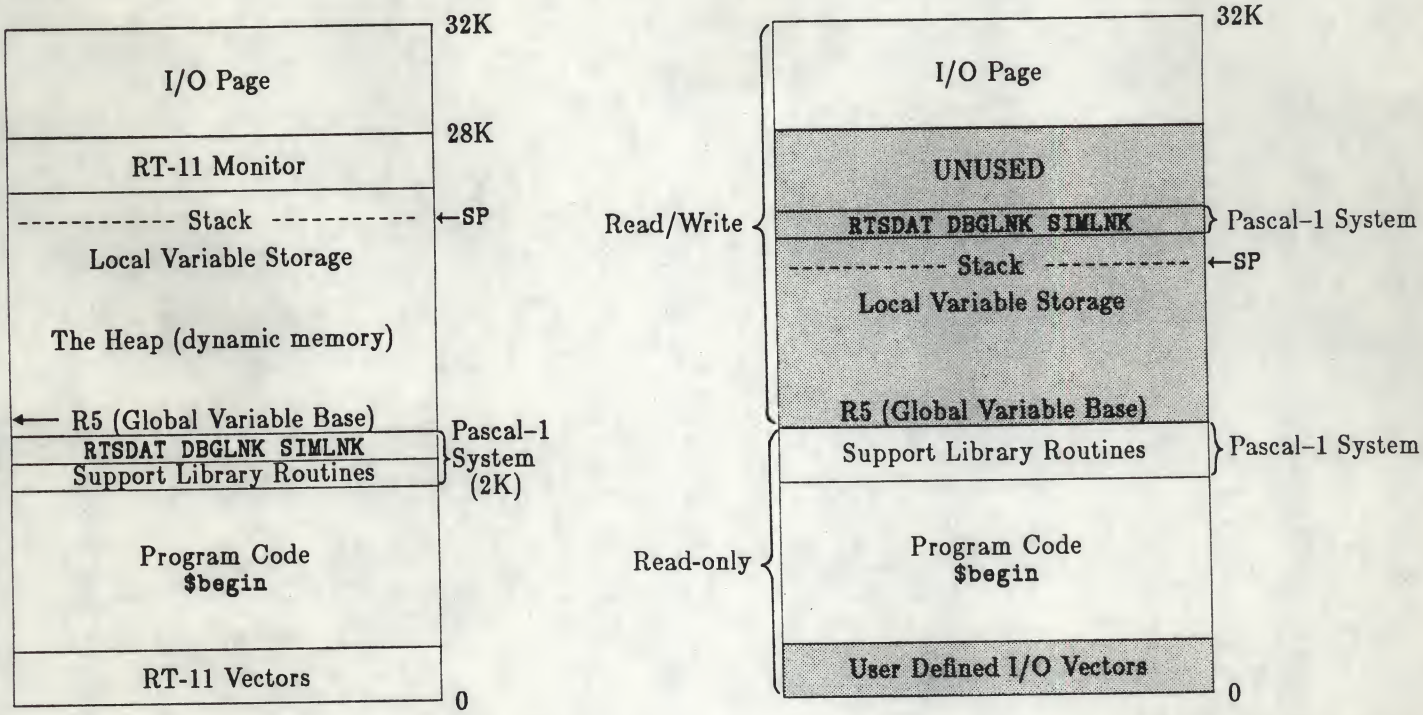
In order to locate the read/write data outside of the ROM area, the run-time system must be modified as shown in Figure 1-B. An obvious change is in the Pascal-1 system partition. The data areas defined as `RTSDAT`, `DBGLNK`, and `SIMLNK` must be moved to read/write memory locations. These are program sections defined and accessed by the Support Library.

`RTSDAT` is a 195-word data area used by many of the Support Library routines. The global variables of importance are:

```
$RESR6 - initial stack value
$RESR5 - initial global area pointer
$KORE  - top of heap pointer
$FREE  - pointer to head of free list.
```

`DBGLNK` is a 2-word data area used for linking to the debugger. `SIMLNK` is a 1-word data area used for linking to the simulated FIS routines.

Continued on Page 8



1-A. Pascal-1 Run-time Memory Organization

1-B. Modified Pascal-1 Run-time Memory Organization

Figure 1: Memory Organization

When building an executable Pascal program, it is the RT-11 linker that loads the Support Library routines (from PASCAL.OBJ), immediately following the user program space. However, in the configuration of Figure 1-B, this will be read-only memory. The "/Q" option of the linker allows you to specify the absolute base address of up to eight program sections ("psects"). Use this option during the link phase to load the RTSDAT psect in a read/write memory location. The psects, DBGLNK and SIMLNK, will follow the relocated RTSDAT area so they need not be explicitly specified to the linker.

Another change implied in Figure 1-B is in the partition allowing for dynamic memory allocation. This partition is bounded by the global variable base (R5) and the stack pointer (SP) and must be located in read/write memory. The routine $START in the Support Library initializes these boundaries; R5 is set to the memory location immediately following the Support Library routines and the SP is set to the top of available memory as shown in Figure 1-A.

$START also initializes the variables located in the RTSDAT data area, sets the trasnfer address to $START, and starts up the user program at $BEGIN. The transfer address is the address at which a program starts execution when running under the control of the RT-11 monitor.

This dynamic memory partition must be reconfigured so that it resides outside of the ROM area. A method of relocating this partition is to write an initialization routine the performs the $START function. This routine should be written in MACRO-11 assembly language, assembled separately, and linked together with the Pascal code modules. The routine must first set the SP to the upper bound and the global variable base (R5) to the lower bound, both being read/write memory locations. Then, it must set $RESR6 to the value of SP and $KORE to the value of R5; zero the 2-word area starting at $FREE, and the area between R5 and SP. Finally, it must do a jump to $BEGIN to start execution of the user's main program block.

The following code is an example of the initialization routine:

```
        .TITLE PSCINI
        ;Pascal-1 Run-Time System Initialization for Downhole Monitor
          (T:HM)
        .GLOBL $BEGIN,$RESR6,$KORE,$FREE
PSCINI::
        .ASECT
        .=0
        ;Initialize any interrupt vectors here
        ;Starting address of code is at location 400.
        .ASECT
        .=400
        ;
        ;Initialization based on 16K DHM data area (Core)
        ;
        MOV $127144,SP  ;Set stack to start-2 of Pascal run-time area
        MOV $100000,R5  ;Set global variable base above DHM code
        ;
        ;Initialize Pascal system global variables
        ;
        MOV SP,$RESR6   ;Save initial stack pointer
        MOV R5,$KORE    ;Bottom of heap
        CLR $FREE       ;Free list
        CLR $FREE+2
        ;
        ;Zero heap to stack
        ;
        MOV R5,R0
        MOV SP,R1
        SUB R0,R1
        ASR R1
        INC R1
1$:     CLR (R0)+
        SOB R1,1$
        ;
        ;Start DHM program
        ;
        JMP $BEGIN
        .ENG PSCINI      ;Sets transfer address to PSCINI
```

Writing the initialization routine in this manner allows you to perform software development using the RT-11 monitor prior to loading the program into ROM.

The Pascal keyword `ORIGIN` is a Pascal–1 extension which allows access to fixed memory addresses. This is useful for associating a variable identifier with a specific memory address, in particular for locating data at a known read/write memory address outside of the dynamic memory partition and for accessing I/O registers in the I/O page.

RT-11 Based Code

Microcomputer-based software systems that are designed

Continued on Page 10

to execute in ROM usually operate in a standalone configuration. Therefore, the services of an operating system are not available. Referring to Figure 1 again, notice the RT-11 monitor and RT-11 vectors, which are components of the operating system, have been omitted in the modified run-time system. Standard I/O device handlers, such as the console terminal, and standard I/O conversion routines provided by the RT-11 monitor are not resident in the system. In addition, those routines in the Pascal Support Library that issue programmed requests cannot be used. Programmed requests are dependent upon the RT-11 operating system. This implies that the Pascal standard procedures GET, PUT, READ, READLN, WRITE, and WRITELN or any Pascal file processing cannot be used.

The I/O handlers, I/O conversion routines, and I/O vector initializations required by interrupt service routines must be explicitly coded in the software. There are software packages available, i.e., SIMRT, that perform RT-11 emulation of programmed requests. They provide the necessary code to execute the RT-11 programmed requests normally available when using the RT-11 monitor. This greatly simplifies the task of writing code to perform standard I/O processing.

For Pascal programs whose flow of control eventually exits from the program, the last instruction in the main program block generated by the Pascal system is a JMP $END. If a Pascal program exits, this instruction is executed. The $END routine in the Support Library executes the programmed request .EXIT, which terminates the user program and returns control to the RT-11 monitor. The obvious solution to this problem is not to exit from the program. Most software systems that are designed to operate in ROM are real-time systems in which the flow of control is continuous so the JMP $END instruction is never executed.

The $START routine in the Support Library also executes several RT-11 programmed requests. For this reason and the reason described in the preceding section, this routine cannot be executed as a part of the normal Pascal system start-up. If an initialization routine is coded as suggested in "Read/Write Data", then $START will not be executed. There may be other routines in the Support Library issuing RT-11 programmed requests. These may be executed when using particular Pascal–1 language constructs. Beware!

Interrupt Service Routines and Vector Initialization

An interrupt service routine transfers data from an external I/O device and may perform calculations with the data. The service routine is executed as the result of a random hardware signal. This hardware signal causes the processor to interrupt the execution of the main program, enter the service routine, and resume execution of the main program upon completion of the service routine. The location of the service routine is defined by the contents of its assigned I/O vector pair. The I/O vector pair is located in low memory which is determined by the hardware. The contents of the I/O vector pair are a function of the software called vector initialization.

Because of the nature of an interrupt service routine, certain precautions must be taken when coding the service routine in Pascal–1. First, on entry to the service routine, you must disable the stack overflow check provided by the Pascal–1 compiler and enable it on exit from the service routine. Besides eliminating a few instructions, this will save on execution time in the service routine. The stack overflow check can be controlled by using the embedded compiler switch $T. Second, the contents of the general registers R0 through R5 must be preserved during the execution of the service routine. The reason is that the Pascal system, as well as the code generated by some Pascal instructions, uses these registers. Upon entry to the service routine, the registers must be saved on the stack (R6) and restored from the stack on exit from the service routine. Third, the register R5 must be set to the value of the system variable $RESR5. $RESR5 contains the initial value of the global area pointer and will initialize R5 properly for accessing global variables within the service routine. Finally, a return from interrupt (RTI) instruction must be executed at the completion of the service routine. This will cause the main program to resume execution.

The Pascal–1 compiler allows MACRO-11 assembly code to be placed at any point in the Pascal program by using the embedded switch $C. Since this feature is an extension of the Pascal language standard and prevents portability of the code to a different processor, it is recommended that all assembly code be isolated in separate procedures or separate source files.

Embedded assembly code must be used in the interrupt service routines for preserving the registers, initializing R5, and returning from the interrupt. The remainder of the code may be written using Pascal instructions. However, for those service routines requiring critical timing, it may be necessary to write the entire body of the service routine in assembly code (i.e., between the BEGIN and END instructions). The MACRO-11 instruction to return from interrupt (RTI) must immediately precede the Pascal END instruction.

Each interrupt service routine should be coded as a single Pascal procedure. All service routines can be placed in a single Pascal external module using the embedded compiler switch $E. A Pascal external module can include global declarations. In fact, if the service routines access global variables declared in the main program module, the global

declarations must be included in the external module exactly as they appear in the main program module. The reason for this is the manner in which the Pascal system accesses global variables using R5 as a base pointer. It is a recommended practice to include all global declarations of types, constants, and variables in a separate source file. This file and the external module are then concatenated during the compile phase and finally linked with the main program module. (NOTE: the file containing the global declarations is also concatenated with the main program during its compile phase.) Placing the service routines in a Pascal external module allows the service routine procedures to be identified to the linker with global entry names. This will be very useful for the vector initialization.

The following code is an example of a Pascal external module containing an interrupt routine which services the event line clock.

```
(* The following global declarations are contained in a separate file
   called DHMGBL.PAS *)
  var
   Clockctr,SECONDS,MINUTES,HOURS : integer;

(* The following external module called HNDLRS.PAS contains the
   interrupt service routines *)
(*$E + Make global level procedures external *)
(*$T - Disable stack overflow check for interrupt routines *)
(*$C
      .ASECT          ;START INTERRUPT ROUTINES AT LOC.500 FOLLOWING THE
      .=320           ;INITIALIZATION ROUTINE AT LOC. 400; NOTE: THE RADIX
                      ;FOR EMBEDDED ASSEMBLY CODE IS DECIMAL; 320 = 500 OCTAL
*)
procedure CLOCK;
  begin (* CLOCK *)
    (*$C
       MOV R0,-(SP)    ;SAVE REGISTERS
       MOV R1,-(SP)
       MOV R2,-(SP)
       MOV R3,-(SP)
       MOV R4,-(SP)
       MOV R5,-(SP)
       MOV $RESR5,R5   ;INITIALIZE GLOBAL BASE POINTER R5
    *)
  ClockCtr := Clocktr + 1;
  if ClockCtr > 50
    then
      begin
        ClockCtr :=0;
        SECOND := SECONDS + 1;
        if SECONDS > 59
          then
            begin
              SECONDS := 0;
              MINUTES := MINUTES + 1;
              if MINUTES > 59
                then
                  begin
                    MINUTES :=0;
                    HOURS := HOURS + 1
                  end
            end
      end;
```

Continued on Page 12

```
(*$C
   MOV (SP)+,R5        ;RESTORE REGISTERS
   MOV (SP)+,R4
   MOV (SP)+,R3
   MOV (SP)+,R2
   MOV (SP)+,R1
   MOV (SP)+,R0
*)
end; (* CLOCK *)

(*$T+ Enable stack overflow check *)
```

The user defined I/O vector partition illustrated in Figure 1-B is a fixed partition ranging from location 0 to location 376. Each I/O vector pair in this partition provides the necessary linkage between the main program module and the interrupt service routine. When generating the absolute load image file (.LDA) for loading into ROM, all I/O vector pairs in this partition should be initialized to handle unexpected traps or interrupts. There are no restrictions on .ASECT directives if the output file format is .LDA.

Since this partition resides in ROM, the vector initialization cannot be performed at run-time but must occur prior to loading the code into ROM. This can be done easily by inserting the appropriate MACRO-11 instructions in the initialization routine (refer to the code example in "Read/Write Data"). To initialize the I/O vector pair for the preceding clock service routine, the following instructions are inserted into the initialization routine:

```
.ASECT
.=100  ;I/O VECTOR PAIR FOR THE CLOCK IS LOCATED AT 100
.WORD CLOCK ;GLOBAL ENTRY ADDRESS DEFINED IN INTERRUPT ROUTINE
.WORD 340 ;VALUE OF THE PROGRAM STATUS WORD (PSW)
```

For software development under the RT-11 operating system, prior to loading into ROM, generate a save image file (.SAV) and initialize only those I/O vector pairs normally required by the application. Vector initialization can be performed automatically in the initialization routine as shown in the preceding code example. However, there are some restrictions on the use of the .ASECT directive for a .SAV file. Locations 54 and 360-377 should not be modified because these locations are used by the RT-11 monitor to load the .SAV file into memory. Some vectors may not get initialized properly because the RT-11 monitor also sets up vectors for its own use. For example, location 100 is set to the address of the RT-11 clock service routine. To ensure the vectors have been initialized properly, load the .SAV file into memory with the GET command and use the console ODT to verify the vector initialization.

Example: Control and Monitor System

At this point, all of these do's and don'ts may seem overwhelming. However, there are only four ideas to remember for developing code that will reside in ROM:

1. Know the system run-time memory organization.
2. Locate read/write data areas outside of ROM.
3. Replace code dependent upon a resident operating system.
4. Observe recommended precautions when coding interrupt software.

In order to illustrate these ideas, the software coded in Pascal–1 for a microcomputer-based system is presented as an example. The code resides in 16K of EPROM and the read/write data is stored in 16K of core memory.

The load map generated from the RT-11 linker using the following command line is shown in Figure 2.

```
R LINK
OHMAU1/Q,LP:= PSCINI,HNDLRS,DHMAU1,PASCAL
Load section:address? RTSDAT:127162
```

The "/Q" link option locates the Pascal system psects RTSDAT, DBGLNK, and SIMLNK in core memory starting at location 127162. The first input module PSCINI is the initialization routine and is the start of program execution as indicated by the transfer address. The global variables $SAVSP, $PWRFL, and $T1REC located in high memory are miscellaneous program variables also defined in PSCINI. The second input module HNDLRS is a Pascal external module consisting of the interrupt service routines from location 500 to 2276. The third input module DHMAU1 is the main Pascal program whose main block starts at $BEGIN. The last input module PASCAL is the Pascal–1 Support Library which contains the routines defined by the global entry

addresses from $START to SIMLNK.

```
RT-11 LINK  V06.01      Load Map        Wed 01-Apr-81 08:12:32
DHMAU1.LDA      Title:  PSCINI  Ident:  V1.26

Section  Addr   Size    Global  Value   Global  Value   Global  Value

  ABS.   000000 002440   (RW,I,GBL,ABS,OVR)
                        $VER    000014  DATACQ  000500  DATXFR  001054
                        TRAP    001102  PWRDUP  001130  CLOCK   001266
                        T1      001446  T2      001656  T3      001740
                        T4      002102  T5      002272  T6      002276
                        $SAVSP  127150  $PWRFL  127152  $T1REC  127154
         002440 124522   (RW,I,LCL,REL,CON)
                        PSCINI  000400  $BEGIN  052730  $START  060526
                        $B63    061462  $END    061462  TSTCHN  061550
                        CLRSCR  061646  $SETIO  061664  $B61    061770
                        $GET    061776  $GETCH  061776  $B60    062212
                        $PUT    062220  NXTBUF  062424  CHKEOF  062734
                        $B68    062764  $CLOSE  063024  $B62    063174
                        $BREAK  063202  LOWSTK  063242  HISTK   063266
                        $ZAPCH  063316  READ    063570  $B50    064444
                        $B40    064646  $B52    064752  $B54    065012
                        $B48    065076  $B56    065130  $B58    065146
                        $B42    065076  $B46    065240  $B44    065302
                        $B116   065330  $B118   065350  $B120   065374
                        $B78    065416  $B80    065444  $B82    065476
                        $B74    065642  $B127   065656  $B76    065674
                        $NOMEN  065712  $B75    065762  $B77    066006
                        $B125   066024  $B70    066070  $NEW    066074
                        $B72    066366  $DISPO  066372  $ERROR  066676
                        ERROR   067314  $B24    070054  $B26    070062
                        $PUTCH  070304  $B20    070356  $B22    070364
                        $FLUSH  070422  $B36    070440  $B38    070446
                        $B32    070476  $B34    070504  $B110   070566
                        $B112   070574
 RTSDAT  127162 000606   (RW,I,GBL,REL,OVR)
                        $RESR6  127162  $RESR5  127164  $OUT    127174
                        $CHANO  127176  $USRPC  127236  $FILE   127242
                        $KORE   127244  $FREE   127246  RTAREA  127314
                        $STACK  127770
 DBGLNK  127770 000004   (RW,I,GBL,REL,OVR)
 SIMLNK  127774 000002   (RW,I,GBL,REL,OVR)
                        $F4     127776  $F8     127776

Transfer address = 000400, High limit = 127774 = 22526. words
```

[ ] Pascal-1 SUPPORT LIBRARY
[ ] USER-CODED ROUTINES

Figure 2. Load map example.

The input modules PSCINI, HNDLRS, and DHMAU1 to the

Continued on Page 14

RT-11 linker are user-coded modules and are built as follows:

```
R PASCAL
HNDLRS = DHMGBL,HNDLRS
DHMAU1 = DHMGBL,DHMAU1

R MACRO
PSCINI = PSCINI
HNDLRS = HNDLRS
DHMAU1 = EMRT,USRMAC,DHMDEF,DHMAU1
```

The file DHMGBL contains all global declarations; EMRI and USRMAC make up an RT-11 emulation package; and the file DHMDEF contains a table of variable length strings defined in MACRO-11 assembly language which are accessed by the main Pascal program.

Bibliography

1. *Digital Microcomputer Processor Handbook*, Digital Equipment Corporation, 1979
2. Pascal–1 *Version 1.2 RT-11*, Oregon Software, January 1981.
3. *RT-11 System User's Guide*, Digital Equipment Corporation AA-5279B-TC, March 1980.
4. Peter Grogono, *Programming in PASCAL*, Addison-Wesley Publishing Co., Don Mills, Ontario, 1980.
5. Kathleen Jensen and Niklaus Wirth, *PASCAL User Manual and Report*, Springer-Verlag, New York, 1979.

Distributors' Seminar

Distributors from around the world joined Oregon Software personnel in mid-May to discuss ways that both can work together to increase sales and provide better support for customers.

The distributors' seminar was part of Oregon Software's fifth annual open house festivities May 17-18. Attending the seminar were Joachim Schmitz of ACCopy (West Germany); Maurice Munsie of Network Computer Services (Australia); Jack Prior and Suzanne Swan of Prior Data Systems (Canada); Gail Thornley of Real Time Products (United Kingdom); Herje Wikegård of Sern Dator Konsulta AB (Sweden); and Steve Brecher of Software Supply (California).

Oregon Software heard a number of concerns from distributors, ranging from technical support to pricing. We gathered feedback that will be useful for long-term planning and answered a number of specific questions for the visitors.

Key topics included:

- Upcoming major revisions to Pascal–1 (V1.3) and Pascal–2 (V2.1). (These changes will be described in the fall issue of the Pascal Newsletter.)
- New products from Oregon Software, including the RSX-to-VERSAdos cross-compiler, a native VERSAdos compiler, and a concurrent programming package that goes with either of them.
- Possible future products.
- Better ways for Oregon Software and distributors to respond to Trouble Reports.
- Software law and licensing agreements and how these affect distributors.
- The pricing of products.

Trouble Reports. Efficiency was the key word in the discussion of how to better handle Trouble Reports. Distributors expressed a desire for for faster turnaround and a more efficient reporting method. We are considering several suggestions, including the possible use of an international networking system to allow distributors to quickly find out what bugs have been fixed and what bugs haven't.

Licensing. Distributors were briefed by Oregon Software's lawyers on the need for strict licensing agreements and learned that we are developing a simpler (but just as strict) agreement for use later this year. We also discussed and clarified the policy for update requests.

Pricing. Oregon Software's OEM sales manager, Paul deBruyn, led a discussion on possible changes in the company's pricing structure. We will consider distributor suggestions in formulating a long-term marketing and sales plan.

In the future, Oregon Software plans more distributor seminars. Next year's may be in conjunction with the 1983 DECUS conference, when many international distributors will be in the U.S.

Exterminated Any Interesting Bugs lately?

We've consulted our libraries and our consultants, but no one can give us good examples of the kinds of things that often go wrong ... no one but *you*. We're thinking of starting a bug collection, of pinning those nasty system-crashing specimens to a board under a glass case, right here at Oregon Software. So keep your nets and formaldehyde ready. You have just a little more than three months to catch the most interesting bug. Prizes, contest rules, and entry deadline will be announced in our next quarterly (no laughing out there!) newsletter.

RSX System Commands from Pascal

The procedure **DoCmd**, described in this article, answers many customers' requests for an easy way to delete, copy, and rename files from within a Pascal program. **DoCmd** enables RSX programs to issue any system command as if the user had typed the command at a terminal. **DoCmd** also serves as an example of ways to process asynchronous system traps (ASTs) with Pascal.

To use **DoCmd**, you must first assemble the MACRO source, DOCMD. MAC. This module defines **DoCmd** as an external Pascal procedure. Pascal–2 and Pascal–1 use **DoCmd** in the same way, but Pascal–1 users must modify the macro source to call the procedure from Pascal–1 programs.

DOCMD.MAC performs three operations: detaches user's terminal, allocates a status return block on the heap for the spawned tasks, and spawns MCR, passing it a command line. Each of these operations is explained in the following paragraphs.

To produce the object file DOCMD.OBJ, you type in the program and assemble it with the PASMAC utility. The first part of the **DoCmd** procedure consists of PASMAC directives to define the procedure and its parameters. The registers used by the procedure are saved on the stack.

All Pascal tasks attach to the user's terminal (logical name `TI:`) when initialized. You can prevent the Pascal support library from attaching to `TI:` by patching the global symbol **$NOATT** to be a NOP instruction (octal 240). To do so, use the GBLPAT option of the Task Builder. If the root segment of a task is TEST (see the Task Builder map), then the patch is:

```
TKB>GBLPAT=TEST:$NOATT:240
```

Oregon Software recommends that Pascal tasks run via the **DoCmd** procedure use this patch to prevent the support library from attaching the terminal, allowing several tasks to use `TI:` at the same time. When **DoCmd** calls the DETACH procedure (defined in the Pascal support library) to detach your task from `TI:`, a task activated (spawned) by **DoCmd** can use the terminal. The logical name `TI:` in the spawned task will be the `TI:` of the task that called **DoCmd**.

After detaching from the terminal, the **DoCmd** procedure allocates a 9-word status return block on the heap. In 8 words of the status block, the system spawn directive returns the status of the activated task. The extra word holds a pointer to the user's status variable. The system spawn directive for RSX version 3.2 runs the MCR and passes it a command line defined by the caller. The specified AST routine will be activated on termination of the spawned task.

When the AST routine is entered, the address of the status return block will be placed on the stack. The AST routine uses the status return block to obtain the address of the user's status word. The status of the spawned task is then returned to the user and the status block is deallocated. Although you could modify **DoCmd** to use an event flag rather than an AST routine, this could limit the number of simultaneous tasks that could be initiated by a Pascal program.

The code for DOCMD.MAC is shown on page 16.

```
.title docmd Execute MCR command line
        .mcall spwn$s,astx$s

        proc    docmd                           ; Execute MCR command
        param   cmd,address                     ; Address of command line
        param   cmdlen,integer                  ; Length of command line
        param   status,address                  ; Address of returned status word
        save    <R0,R1,R2,R3>                   ; Save registers
        begin
        jsr     PC,detach                       ; Detach from TI:
        mov     cmd(SP),R0                      ; Point to command line
        mov     cmdlen(SP),R1                   ; Get command length
        mov     status(SP),R2                   ; Point to status word
        clr     (R2)                            ; Set to zero to show not completed
        mov     #18.,-(SP)                      ; Ask for 9 word status block
        jsr     PC,$new                         ; Allocate dynamic memory
        mov     (SP)+,R3                        ; Point to allocated memory
        mov     R2,(R3)+                        ; First word is pointer to user status
        spwn$s #mcr,,,,,,#cmdone,r3,r0,r1       ; Spawn MCR
        cmpb    #is.suc,$dsw                    ; Was directive accepted?
        beq     10$                             ; Yes
        mov     $dsw,(R2)                       ; No, give error
10$:    endpr                                   ; Return to caller
;
;
; AST routine to send returned status to user
;
cmdone: sub     #2,(SP)                         ; Point to start of status block
        mov     R0,-(SP)                        ; Save R0
        mov     2(SP),R0                        ; Point to status block
        mov     2(R0),@0(R0)                    ; Send offspring status to user
        mov     R0,-(SP)                        ; Push address to dispose of
        mov     #18.,R0                         ; Size of block
        jsr     PC,$dispose                     ; Dispose of status block
        mov     (SP)+,R0                        ; Restore R0
        tst     (SP)+                           ; Kill status block address
        astx$s                                  ; Return from AST
mcr:    .rad50  /MCR.../                        ; Name of MCR
        .end
```

Once the object file is assembled, the **DoCmd** procedure can be called from a Pascal program. The sample program, CMDTST.PAS, shows you how RSX command lines might be might be issued from within a Pascal task. The program consists of two external procedures and the main program.

To issue the command line, the procedure **DoCmd** is passed three parameters. The first parameter, the address of the command line, is passed by reference (**var** keyword). The maximum length of a command line is 79 characters. Since only the address of the command line is passed to the **DoCmd** procedure, you may wish to use a shorter maximum command line length, especially when using literal command strings. The second parameter is an integer value specifying the number of valid characters in the command line. This number must lie in the range 1..79. In the sample program, the number of valid characters is determined by deletion of the trailing spaces in the command line. The third parameter, the status word, is passed by reference. This integer variable is set to zero when the command is sent to the MCR. When the command is completed, the status word will be modified to contain the status returned by the command. Typical status returns are:

```
1 -- Success
2 -- Error
4 -- Severe error
```

Other status returns are possible. The procedure **exitst** is defined in the Pascal support library as an external Pascal

procedure accepting one integer parameter. This procedure will terminate a Pascal task and return a 16-bit status value. If the spawn directive is rejected, the status word will contain the directive status word (DSW) code.

The FORTRAN interface to the system routine, procedure wfsne, waits for a significant event, such as the execution of an AST routine. A loop in the main program allows **wsfne** to continue waiting for significant events until the status word returns a non-zero value. This loop must be used because other tasks on an RSX system can cause significant events. While CMDTST loops, each command line typed by the user is passed to the MCR, and when the command is completed, the status word is printed. When a command line is null, CMDTST exits the loop.

```
program cmdtst;
  type
    Command_Line = packed array [1..79] of char;
  var
    Cmd: Command_Line;
    Cmdlen: integer;
    Status: integer;

  procedure DoCmd(var Cmd: Command_Line;
                  Cmdlen: integer;
                  var Status: integer);
    external;

  procedure WFSNE;
    nonpascal;

  begin
    write('command? ');
    readln(Cmd);
    Cmdlen := 79;
    while (Cmd[Cmdlen] = ' ') and (Cmdlen > 1) do
      Cmdlen := Cmdlen - 1;
    DoCmd(Cmd, Cmdlen, Status);
    while Status = 0 do
      WFSNE;
    writeln('status=', Status: 1, '.');
  end.
```

The **DoCmd** procedure is a powerful tool limited only by your imagination. It gives a Pascal program the ability to perform any operation you could do at your terminal. For example, the sample program could be modified to initiate several tasks simultaneously by using a different status word for each task.

Besides issuing MCR commands, **DoCmd** can perform tasks such as:

- Run PIP to rename, delete, copy, or purge files.
- Have the Pascal program create and close a command file (say, TEST.CMD) which can then be passed to the indirect command file processor (@TEST).
- Send a series of commands to allocate a floppy drive, search for bad blocks, initialize, and mount a floppy.
- Run PIP to get a directory listing of some UIC and write it to a file. Then open that file in a Pascal program and perform some operation on each file.
- Run the Pascal compiler; and, if the compilation contains no errors (status word=1), run the Task Builder.
- Change the system date, create or delete partitions, set terminal characteristics.

This procedure can even log out!

RT-11 File Variable

RT or TSX users often ask us for help with record-locking or retrieval of channel. A representation of the RT-11 file variable can be included in a Pascal program to allow the user access to the file variable created by the support library.

The short program code below demonstrates the use of the RT-11 file variable to retrieve the current block and channel numbers for an open file. Then the program seeks a record (record locking is only found under TSX) that is not locked by another user. If the record is locked, the current block pointer in the file variable will indicate to the seek procedure that the record is already in the buffer. So, the program must force the buffer to reread the disk on each try.

```
program Rtfilevar(input,output)      {The RT-11 File variable}

   {Oregon Software reserves the right to change the internal structure
    of file variables in future revisions of Pascal-2.}

type
  Byte = 0..255;
  Rad50 = 0..65535;
  StatusBits = (MustWrite, Spanned, SeekOK, DoRead, DoWrite, PendingEOF,
                TempFile, TextFile, NonFileStructured, Go, Defined, Wait,
                ODTMode, Dirty, EolnFlag, EofFlag);
  DeviceBits = (NotUsed, AlsoNoUse, SpecialDev, HandlerDie, NonRTDir,
        WriteOnlyDev, ReadOnlyDev, RandomAccess);
  HandlerIndex = (Blank, Blank1, TTYHand, Blank2, Blank3, Blank4, Blank5,
                  Blank6);

  FileBuffer = packed array [0..511] of byte;

  FileBlock = packed record
    BuffPointer : ^FileBuffer;        {Where the data is}
    Status : set of StatusBits;
    Device : Rad50;                   {Name of logical device}
    Name1, Name2, Extension: Rad50;   {FileName and extension}
    FileSize : integer;               {Number of Blocks total}
    Channel: Byte;                   {RT-11 Channel Number}
    Function : Byte;                 {Reserved and not used}
    Block: integer;                  {Current disk block in buffer}
    BufferAddress: integer;          {Where the buffer is}
    BufferSize : integer;            {How large}
    Handler : set of HandlerIndex;
    DeviceType: set of DeviceBits;
    RecordSize : integer;            {How large is each individual record}
    EndOfDataPointer: integer;       {Points to end of record in buffer}
    RecordsPerBlock: integer;        {Should equal 512 div recordsize}
  end;
```

```
FileBlockPtr = ^FileBlock;

var
  f: text;

  function CurrentBlock(f:FileBlockPtr):integer;

    begin
      CurrentBlock := f^.Block;
    end;

  function CurrentChannel(f:FileBlockPtr):byte;

    begin
      CurrentChannel := f^.Channel;
    end;

  procedure ForceBufferRead(f:FileBlockPtr);

    begin
      f^.Block := -1 ; {Tells the support library to re-read the disk}
    end;

begin
reset(f,'TEST.DAT/SEEK');
{First get a block number}
writeln(CurrentBlock(loophole(FileBlockPtr,f)):1);
{Now write the current channel number}
writeln(CurrentChannel(loophole(FileBlockPtr,f)):1);
{Now go get record 10 and thrash the disk until we get it}
repeat
  ForceBufferRead(loophole(FileBlockPtr,f));
  seek(f,10);
until RecordIsNotLocked;
end.
```

The Log: Pascal-1 and Pascal-2

Oregon Software's first three Pascal Newsletters described the significant changes in Pascal–1 V1.2 from release V1.2A in December 1979 through V1.2J in December 1981. This issue describes the significant changes from V1.2J through the present release, V1.2K.

This log also describes the significant changes in Pascal-2 since the release of V2.0J in December 1981 through the present release, V2.0K.

To use this log, first determine what release of V1.2 or V2.0 you have. (The release number is printed on the headline of all program listings.) Then review the log to determine the changes made since the release of your version. If the changes are of particular importance to your application, your Designated Contact Person should request an update, in writing. The DCP is usually the senior programmer in charge of software. You will receive the latest version of the software.

The first section of this log describes changes applicable to Pascal–1 for all operating systems. The next section describes Pascal–2 changes for all operating systems.

Future issues of the newsletter will continue to outline improvements to Oregon Software's Pascal products.

Pascal-1

V1.2K includes these changes.

Incorrect evaluation of set expressions

Compiler generated error messages for legal Pascal expres-

sions involving **in** operands when the order of operands was reversed.

Odd address trap errors

In certain cases, an illegal identifier produced an odd address trap.

'With' statement record pointer clobbered

Unpredictable results occurred when a procedure, passed as a parameter, was called from within a **with** statement.

Variant tag list bug

Compiler has been modified to accept tag fields.

'For'-loop iteration problem

For loops were being executed more times than specified.

Pascal-2

V2.0K corrected these problems

'Nonpascal' definition level changed

Nonpascal procedures could be defined at any level. Now, **nonpascal** procedures must be defined at the global level.

Packed record and packed array problems

Problems were corrected in the allocation of packed records and packed array indices.

Multiple 'goto' problem

Multiple **gotos** to the same label incorrectly caused an error if the **gotos** were nested.

'Origin' variable incorrectly

In some uses, **origin** variables were assigned to the wrong address.

Extra semicolon causes problems

An extra semicolon before an **else** clause caused a reserved instruction trap instead of an error message.

Compile-time odd address trap errors

A procedure declared **forward**, then passed as an argument, produced an odd address trap. Also, a string of the form **['a'..z']**, which is missing a single quote, caused an odd address trap instead of the appropriate error message. Third, some 'ord' expressions were evaluated incorrectly, causing an odd address trap.

Some set operations calculated incorrectly

A set intersection operation produced a result that contained far more elements than actually belonged to the intersection.

Bad code generated for packed, unsigned integers

Packed, unsigned fields yielded incorrect division and comparison results after being unpacked.

Problem with unsigned integers

Incorrect answers resulted when unsigned integers were used in place of signed values.

SQR argument problem

In some situations, **sqr** calculated incorrect results for **/fis/eis/sim** code generation.

More 'undeleted temps'

Internal compiler errors generate this message. In some cases, the compiler failed to delete temporary variables (usually generated during optimizations); in others, the compiler lost track of the stack.

Wrong result with scalar indexing

In some cases, the wrong element of an array was accessed when scalar indexing was being used.

Debugging of 'origin' variables

Origin variables were incorrectly allocated to registers in the same manner as other variables. The Debugger now differentiates **origin** variables from normal variables.

Debugger executed dead code

The "dead-code elimination" optimization was left on dur-

ing debugging, causing the Debugger to lose count of line numbers. Dead code is no longer eliminated when debugging is enabled.

Debugger error writing variables with long names

When the Debugger tried to write variables with names 32 characters or longer, the error "not a valid identifier" resulted.

Incorrect calculation of function values

Some functions with expressions as arguments gave incorrect results.

Error checking fixed

The assignment of `true` to a `real` variable caused an odd address trap instead of producing the appropriate error message.

Prose feature fixed

The Prose "all upper-case" feature, `.option (u+)`, now works properly.

'With' statement erred

In some situations, the `with` statement generated code that referenced the wrong record field.

Quirks, Bugs, and Limitations

Pascal–2, Version 2.0K for RT-11, has some bugs and/or limitations that are scheduled to be fixed in Pascal–1 V1.3 and Pascal–2 V2.1. Meanwhile, we offer these descriptions of RT-11 problems so you'll know how to avoid or work around them.

BEWARE...

The SJ compiler requires a large amount of memory and must be run using Digital's standard distribution of RT-11. If it is not possible to run the compiler under SJ, use RESORC.SAV (the `show c` command) to check the size of the monitor. The resident monitor base should be around 146574 octal; it cannot operate in significantly less space.

It is still not possible to call FORTRAN IV routines that perform I/O operations from Pascal. FORTRAN IV initializes memory in the same manner as the Pascal support library, causing conflicts in allocation. Also, the FORTRAN interface requires word-aligned variable parameters. So, you may use `nonpascal` to access FORTRAN routines, do everything correctly, and still get odd address traps in the FORTRAN routines. The problem is that FORTRAN expects the `var` parameters to be word-aligned, and Pascal–2 does not align everything on word boundaries. (Pascal–1 does, so no problem there.) For now, you will have to fudge the program's data structures to force the data you plan to pass to FORTRAN routines onto the word boundary.

Users with version 6.01F or greater of LINK.SAV should not have to set the transfer address for all programs created by Pascal. Previous versions of the Linker still contain a bug that will not properly set the transfer address that comes from a module in a library.

When using Pascal–1 or Pascal–2 in a graphics application or any application that does heavy I/O to a text file (TT:, LP:) the program will fail with a "File OverFlow" error after writing 65K blocks of data. The support library increments a counter in the file variable and only checks for the carry bit set regardless of the device it is writing to. A temporary solution is to declare a Pascal representation of the file variable (shown in a preceeding article) and frequently zero `f^.block`.

VICIOUS COMBINATION...

Users of PDP-11/23 processors have reported the error message "Seek on Sequential file 80:8080.80". This message occurs immediately after the system has been booted, usually during a second compilation via the `R PASCAL` command, of a program with errors. The problem results from a combination of the 11/23 memory self-testing during boot, the way the `R` command loads a program, and certain pointers left in memory from a previous compile. The solution is to use the `RUN PASCAL` command instead. Do not keep trying to compile with `R PASCAL`; the bug may eat the monitor on the third or fourth attempt.

TRAPS...

Both the SJ and XM compilers require about 600 blocks of contiguous disk space on device DK:. A bug in error reporting will cause the compiler to trap if enough continuous disk space is not available.

The XM compiler will trap during compilation if a value of "0.0" is passed to a procedure or function with a value parameter of type `real`. The solution is to assign a variable the value of zero and pass the variable to the procedure.

Some syntax errors cause a trap to 4 under TSX-Plus

and not under SJ. One known cause is supplying a constant to the 4th parameter to rewrite or reset, e.g., `rewrite(f,'dk:',filename,3)`.

DON'T...

Two Control-C's will not abort the Debugger. Hitting Control-C twice will interrupt execution of your program and return control to the Debugger. You should then issue the `Q` command to exit to RT-11 (Control-Z for Pascal-1). Some known conflicts with the USR may require a re-boot of the system after double Control-C.

Pascal's initialization routine ($START) does not ensure that you have enough memory to run large programs. Therefore, it is possible for the $START routine to set the initial stack pointer somewhere in the user code on extremely large programs (especially if USR is NOSWAP, extra drivers are loaded, and a large monitor is used). The results are fatal.

DO...

When compiling programs with Pascal-2 always be sure that the USR is SET SWAP.

Errors, additions to manuals

All Pascal-2 manuals

Language Specification, "Mod" of Negative Numbers

The expression for generating the standard result should be:

```
if result < 0 then result := result + j;
```

Introduction

The address for the Pascal Users' Group has been changed to:

```
Pascal Users' Group
Attn:  Rick Shaw
P.O. Box 4406
Allentown PA 18104-4406
```

Pascal-2 RT-11 manual

Option directive's left and right justify switches, page 143

The first example is missing the directive escape character. The correct directive is:

```
.OPTION( L+ R+ ) :
```

Pascal calls FORTRAN

When you use the **nonpascal** directive to call a FORTRAN subroutine, the compiler generates a special kind of parameter list. Using a standard procedure-calling mechanism, this list passes parameters to the FORTRAN routine. This is the only support the Pascal compiler provides for calling FORTRAN.

As some users have found, this does not mean that all FORTRAN routines called from Pascal will operate correctly. You must also initialize the FORTRAN object library before calling FORTRAN procedures from a Pascal program.

Often a FORTRAN subroutine calls other subroutines in the FORTRAN object library, just as a compiled Pascal program calls on routines in the Pascal support library. These FORTRAN subroutines assume that they have been called from a FORTRAN main program. The FORTRAN object library must be properly initialized or calling FORTRAN routines from Pascal can cause traps or incorrect results. This is not a problem unique to Pascal. The same initialization is required for FORTRAN subroutines called from MACRO.

On RSX, calling FORTRAN IV-PLUS subroutines from Pascal sometimes generates Task Builder error messages.

The Task Builder knows the symbol **nluns** as a special symbol with magical properties. When you build a task containing the label **nluns**, the Task Builder assigns to that location a value equal to the number of logical units specified by the UNITS option. The default value is six. When a Pascal program calls a FORTRAN subroutine that opens a file or does I/O, both the Pascal support library and the FORTRAN object library need to know how many logical unit numbers are available for open files. The Task Builder, being a very clever program, complains about the double definition of **nluns**.

On RSX systems, we have discovered a way to work around these initialization problems, using a small MACRO program to initialize the FORTRAN object library. Initialization takes these four steps:

1. Create the following short Macro program:

```
.TITLE FORINI
        FORINI:: MOV .NLUNS,.MOLUN-2
                 JSR PC,OTI$
                 RTS PC
```

Compile this small MACRO routine and use the librarian (LBR) to insert it into the Pascal support library.

2. In each Pascal program that calls FORTRAN subroutines, define the following external procedure:

```
procedure Forini; external;
```

3. After the **begin** statement marking the start of your main program, call the Forini procedure:

```
begin
  Forini;
  rest of program
end.
```

When you task-build the program, the Forini routine will automatically be loaded from the Pascal support library. You will still get an error for the multiple definition of the symbol **nluns**, but that error can be ignored. The first statement of the Forini routine fills in the value of **nluns** in the FORTRAN object library. The second statement calls the FORTRAN initialization code. Once Forini has been called, you should be able to call almost any FORTRAN subprogram as well as the FORTRAN interface to system directives.

We are still working on the problems of calling FORTRAN IV on RSX and on ways to initialize FORTRAN on RT-11 and RSTS. This will be the topic of a future newsletter article.

Letters

To the Editor:

The new Pascal–2 compiler and run-time system have been in use at FERMILAB for several months, and we are now in a position to comment on the basis of that experience. In general the reactions of the users have been quite positive. We have yet to encounter an operational fault. For a new system, that is unusual and I congratulate Oregon Software for marketing a remarkably solid and reliable product.

Most of our Pascal programs communicate with and make extensive use of the resources of the RSX-11M operating system. As a consequence, the extensions in Pascal–1 and Pascal–2 are used frequently (the changes in the style of the extensions has been a major contributor to our conversion problem). The loss of the "code insertion" feature of Pascal–1 caused some difficulties at first; however, the conversion to external procedures for those insertions which were not covered by the **packed** attribute or the **loophole** function has led to programs with better structure. The PASMAC macro package was a significant aid in converting the insertions which had to be extracted.

My principal complaint concerns the loss of the **exit** statement. We used it extensively to terminate the normal sequential flow within a program block when abnormal conditions were encountered. This particular problem has now had to be solved by either increasing the nesting depth of **if** statements or, where this became intolerable, by use of the **goto** statement. In either case, both the structure and readability of the programs has suffered, and procedures that were once well-ordered and easily understandable have now become noticeably more difficult to comprehend.

I fail to appreciate why its removal was necessary. The fact that it was an "extension" beyond the Pascal standard is no reason when you consider that **origin**, **loophole**, structured constants, and boolean operations upon integers are all extensions and are present in Pascal–2. The **$standard** compiler switch, which facilitates adherence to the Pascal standard, is available to insure portability when such guaranties are required. Nor do I believe that the presence of the **exit** statement presented significant problems in an optimizing, multi-pass compiler. The **goto** statement is at least a problem of the same order and the restoration of context at the **goto** target label is a problem which is similar to that at the exit of a repetition block (**for**, **while**, and **repeat**).

In short, I don't appreciate why it had to be eliminated and, in view of its value in solving the "graceful exit with abnormal conditions" problem, I find the loss exasperating. I would point out, as a final comment on the issue, that many languages which have derived from Pascal, ADA and PRAXIS for example, have statements of this type. In my opinion, future modifications to the Pascal standard will probably include something like the **exit** statement.

A second change which seems unnecessary is the new restriction upon the **origin** attribute of a variable. It can now be used only in the virtual address range of the I/O page and so-called "system space" (addresses at or below 1000 base 8). In the context of the RSX-11M operating system, the **origin** attribute provided a means of referencing shared-common data partitions to which the program was mapped. While it is possible to replace this use of **origin** with a pointer variable, the compiled program will execute with less efficiency because of the additional level of pointer dereferencing.

I hope that these comments are constructive and that they will be of use in future plans for Pascal–2. In the event that any of the points require further clarification, please feel free to contact me.

J. Frederick Bartlett
Software Support Group Leader
Research Services Department
Fermi National Accelerator Laboratory

Editor's Reply:

We have tried hard to adhere to the ISO standard and to encourage Pascal users to do so. As you have pointed out, we have added non-standard extensions, but we have done so only when an extension met one of two criteria. First, the extension must enable users to write well-formed programs in Pascal–2 that would have been impossible in standard Pascal; second, the extension must show a high probability of becoming part of the standard in the near future. We maintain this position out of a desire to save our Pascal users from rewriting serviceable programs when they move them to other standard Pascal compilers. Standardization makes the programs portable, independent of the restrictions of various operating systems and machine implementations.

We see the `exit` statement as syntactically equivalent to a restricted `goto` statement. Both statements are useful when "the natural structure of an algorithm has to be broken", as in the case of the abnormal conditions within a program block that you mentioned. Unlike the `exit` statement, however, `goto` requires a label. We feel that the increased clarity of the `goto` statement is worth the few extra lines of code required for its use. Because the `exit` statement doesn't really add to the language and the ISO committee hasn't shown any disposition toward including it, we don't share your expectation that the `exit` statement will become part of the Pascal standard.

We restricted `origin`'s range to save space in memory. In the context of the RSX operating system, restricting the `origin` attribute of a variable makes less sense than it does in the RT-11 user's environment. Pascal–2 is a very large compiler; as it is now, Pascal–2 barely fits into the available program space of the RT-11 operating system.

Restricting the range of `origin` allows us to simplify certain portions of the compiler. We can keep extra implicit information in the same word as that normally used to specify a relative address, and few changes are required to the hundreds of internal routines dealing with address expressions. The changes required to make `origin` work for any address would make it impossible for the compiler to run on stand-alone RT-11 systems with only 28K words of user memory. The monitor takes about 2K words, leaving the compiler itself with only 26K words of memory available for code. Since small RT-11 systems benefit most from the code optimizations performed by the compiler, we restricted `origin` rather than remove it completely or increase the size of the compiler so that it would not run under RT-11.

As you mentioned, there are alternatives to the `origin` statement. For example, you can force Pascal–2 to reference absolute locations in your address space, using the `loophole` function. This alternative should be as efficient as `origin` because common sub-expression optimization in the compiler produces equivalent code.

We are sorry that these changes caused you such problems, and we take your comments as constructive criticism. We'd be very interested in seeing specific examples of the programs where our alternatives are insufficient.

RSTS/E V7.1 Users' Note

After one minor change to the Pascal–2 CCL interface program, the Pascal–2 system is fully compatible with the lastest release of RSTS/E (version 7.1).

No other incompatibilities have been reported with Pascal–2; none at all have been reported with Pascal–1.

Pascal
NEWSLETTER
Oregon Software, Inc.
2340 SW Canyon Road
Portland, Oregon 97202

Collins Hemingway, editor
David Spencer and Michael Kuhn, staff writers

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. Fee: $100 per newsletter page. Payment upon publication.

Printed in USA

Pascal
NEWSLETTER

NUMBER 5 OREGON SOFTWARE NOVEMBER 1982

SourceTools ready for field test

SourceTools is designed to manage program development in situations where a wide range of products requires numerous source files, where programmers are engaged in cross-system software development, or where several programmers may be working on the same program modules.

Field test will be conducted on the RSX and VMS (compatibility mode) operating systems. Our field-test price for SourceTools will be $2,700 — at least 25% lower than the end-user price. Make-up of the end-user product, and its release date, depends upon field-test results.

We developed SourceTools originally for our own internal use, to manage cross-development projects and update multi-version software, such as our Pascal–2 compilers and cross-compilers for various machines and operating systems. Extensive in-house use has convinced us that these tools can benefit other software shops.

Features

Added Control. When more than one person must obtain copies of a file or alter a program, SourceTools prevents independent authors from making simultaneous and conflicting changes.

A Useful History of Changes. For each change made to a source file under its control, SourceTools records who had it, what changes were made, and why.

Economy of Space. SourceTools stores only the original version of a file and the differences between the source and each successive version, instead of storing each version as a whole copy.

Economy of Time. During cross-development of software, sources are simultaneously maintained on both the host and the target systems. You can reduce transfer time by shipping only the differences between systems.

Efficiency and Accuracy In Re-Creating Software. One SourceTools program mechanizes the re-creation of software from its component modules and codifies the process of updating files. This assures developers that modules are never forgotten, that out-of-date modules are never accidentally incorporated, and that commands are never executed unless they are necessary. Executing a minimum number of commands is more efficient than using command files.

Contents

The SourceTools package contains three groups of programs that work together as a package. The first group, Source Control (SourceCon), consists of four programs for controlling the creation and modification of source files. The individual programs provide these services:

- `NewSrc` creates a new module from a starting file, establishing control information and a "checksum" to ensure validity of updates or changes.
- `GetSrc` provides a user with a version of the source file that can be used or modified. When a source file is checked out for modification, GetSrc denies subsequent requests to modify the source, providing copies only, until the file is returned (see UpdSrc). By default, GetSrc retrieves the most current version, but any earlier version can be specified.
- `UpdSrc` re-installs as the current source an edited version of a copy that was checked out with GetSrc. UpdSrc records controlling information about the update (see PrtSrc), and automatically determines the differences between the current version and the file that was checked out originally.
- `PrtSrc` prints the stored information about the differences between any two versions of a module, including the name of the author, the time of the change, the author's written description of the changes, and the sequence of editing commands needed for conversion to a later version.

Using SourceCon programs, you can access all versions, and you can trace the history of all changes.

Continued on Page 2

In this issue...

SourceTools Ready . . . . . Page 1
Field test policy . . . . . Page 2
OPUS Communique . . . . . Page 2
Information exchange . . . . . Page 3
Letters . . . . . Page 3
Pascal programs as RT-11 System Jobs . . . . Page 4
Reserving room for RT-11's USR . . . . . Page 5
Reducing a program's task image size . . . . . Page 9
Literal strings in structured constants . . . . . Page 8
The Log: Pascal-1 and Pascal-2 . . . . . Page 8
Bug contest begins . . . . . Page 9
Customer survey . . . . . Page 11

SourceTools Continued from Page 1

The second group contains the two programs, TextCom and SEdit. TextCom can generate a script file for parallel source maintenance on different processors. Using the same efficient algorithm used in the SourceCon programs, TextCom compares two files and lists the differences between them. A command-line switch creates an editor script as output for use by a "stream editor" program called SEdit. As a file is read in, SEdit applies the editing commands contained in the script file to a designated file. In the cross-development environment, these two programs work together to minimize host-to-target transfer time. When transferring code from the host development system to the target system, you only ship a short editor script file between systems instead of transferring complete, often lengthy, source files.

TextCom may, of course, be used for any operation that involves text comparison; its use is not restricted to cross-development or generation of editor scripts. TextCom can make better comparisons between files with numerous differences than Digital Equipment's comparison program, DIFFERENCES.

The third component is a single program called Make, which automatically keeps files up to date as components are changed. The user supplies a description file containing the names of files that depend on others and the commands for rebuilding any given file. Make examines the dates of all related files, redoing any out-of-date file, according to the directions in the description file. The Make program executes the fewest number of commands needed to update the modules used in a given file.

Persons interested in becoming SourceTools field-test users should contact Pat Rau at Oregon Software for licensing details.

Field test policy

Oregon Software will be offering its new products to field-test sites for at least a 25%discount off the anticipated end-user price. In exchange for that, we are looking for users who have a thorough knowledge of the hardware/software system on which these programs are implemented and who can evaluate the performance of the new software and suggest ways it may be improved.

Field-test users trade off the hassles of testing software against the benefits of having the product early, getting it at a reduced price, and helping to determine the nature of the final product.

Test-site programmers must be able to cope with "untested" software, to reduce problems to simple examples of one page, and to resolve problems over the phone. Field-test sites must be willing to use preliminary documentation.

Field-test sites will receive the end-user release automatically. The granting of a field-test license, however, does not guarantee that Oregon Software will release an end-user product.

OPUS communiqúe

Oregon Pascal Users Society

Starting with this issue of the newsletter, the newly formed Oregon Pascal Users Society (OPUS) will submit a column dedicated to the sharing of information between Oregon Software and its customers. OPUS is not affiliated with Oregon Software, though Oregon Software is encouraging the group's formation. Membership is free, so join now! Here's the address:

Oregon Pascal Users Society (OPUS)
Bruce Williams
c/o EOCOM
15771 Redhill Ave.
Tustin, California 92680
(714) 730-5051, ext. 302

The need for a society for Oregon Software Pascal users has been evident for a few years. OPUS is an independent organization whose goals are:

1. To share experience in methodology of software engineering, software quality assurance, and/or details of implementation using Oregon Software's Pascal compilers as well as other languages that interface to the Oregon Software Pascal compilers;
2. To disseminate this information through available, expedient and economical channels;
3. Thereby providing a communications channel through which Oregon Software, members of OPUS, and members of external organizations can communicate approaches, techniques, problems, tricks and standards that may speed project development involving Oregon Software's Pascal compilers.

OPUS is being formed by a company in southern California and membership requests have come from as far away as Tasmania. What does it take to become a member? A postcard, letter or phone call.

So far, OPUS has helped users of Oregon Pascal (versions 1.1, 1.2, 2.0 for RT, RSX, RSTS/E) to achieve things as simple as character I/O using RT-11 V2.0 and as complex as AST and event-flag-controlled routines through FORTRAN-called procedures under RSX. If you don't have the NIH ("not invented here") problem, if you think you have something to share or a need someone may have already met, then call or drop a note to OPUS. That will automatically make you a member.

Future communiqués will describe what OPUS members have been working on and report their successes/failures in reaching their goals.

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the Information Exchange. Your description should follow the format of the items below. Interested parties can contact one another directly.

MEASURE, a utility program for use with Pascal–2 under the RT-11 operating system, measures execution time of procedures, functions, and statements, producing a profile of the program in terms of the time spent in execution. E. J. Sauter, Processing Concepts Limited, 2909 North Sheridan Road, Chicago IL 60657, (312) 883-1444.

Pascal Programmers sought by Fortune 500 company with a central data center and large on-line systems. The company is dedicated to using Pascal as a second language; assembler is now the primary language. Write: Information Industries, Inc., 1500 Charter Bank Center, Kansas City MO 64108, Attention: Larry McWilliams.

Software training specialist needed to conduct training classes for customers and employees. Subjects: programming in Pascal language, writing process-control programs, use of RT-11 operating system and utilities, supporting software for ESI laser-processing systems. Contact: Harve Howard, Electro Scientific Industries, Inc., 13900 NW Science Park Drive, Portland OR 97229, (503) 641-4141.

Letters

To the Editor:

Occasionally one reads something which compels one to write a letter of clarification. Such an item is your statement (on page one of the August 1982 issue) that you are releasing the MC68000 OEM cross-compiler now because that processor "does not yet have a good development environment." Your readers should be aware that this statement is factually incorrect.

Despite its potential difficulties as an end-user environment, UNIX is widely acknowledged as a superior development environment. Unisoft of Berkeley has implemented a full native-mode UNIX on the MC68000, with the standard tools from Bell Labs' V7 UNIX as well as most of the University of Berkeley enhancements.

The Unisoft UNIX is available now with the Dual Systems Control Corporation (also in Berkeley) System 83, an IEEE-696 standard (S-100) MC68000 system. Unisoft UNIX is also apparently available for the Sun Workstation and the WICAT, both MC68000-based small systems, and is available to OEMs for inclusion in OEM packages. There are other UNIX versions for the 68000 as well, but I am most familiar with this particular one.

There is at least one "good development environment" for the MC68000.

Ian F. Darwin
Software Supervisor
Communications and Small Systems
University of Toronto Computing Systems

Editor's Reply:

In general, we agree with you that a genuine UNIX system would be a good development environment for the MC68000. The problem is finding one.

Eighteen months ago, when we began our MC68000 project, our statement was definitely true: no good 68000-based UNIX system existed. So we used the mature RSX-11 system to develop the cross-compiler. That remains a fine choice for PDP-based users.

Today, a bewildering array of MC68000-based "UNIX" systems is being promoted. These are similar to UNIX, or based on it, but not identical. Worse, many of them aren't really available yet. We are still monitoring the situation, looking for a UNIX system to use ourselves.

We are aware of Unisoft's UNIX. We did not use that system because it is available for a limited set of configurations, which does not include the EXORmacs. Also, running it requires a great deal of disk space. The system is good, as you say, but of limited availability.

We intend to support a UNIX system soon, but not until we see which of the many contenders becomes the established MC68000 development system.

Editor's Note:

We want to keep our mailing list for the newsletter current. So, please, when you move, let us know where to send your newsletter or ask us to cancel your subscription. To request additional copies or give us the name of a new recipient, contact the editor. Thanks.

Pascal programs as RT-11 system jobs

Mr. Sauter is with Processing Concepts Ltd., 2909 North Sheridan Road, Chicago, Illinois, where he is engaged in systems development using Pascal. His current project is a Relational Data Base Management System (RDBMS) using Pascal as the host language.

Although RT-11 Version 4 is nicknamed "Mini-Tasker", it has only one SYSGEN option, **QUEUE**, to support system jobs, and Digital Equipment Corporation does not recommend the use of other available job slots. For those who are tempted, as we are, to create system jobs, the XM monitor has two important features: virtual overlays and virtual jobs. Assuming you are familiar with the techniques for creating overlaid programs (both virtual and disk-based) explained in the Pascal–2 *Programmer's Guide*, Version 2.0 for RT-11, this article explains how we minimized the low-memory static window size to allow several Pascal programs to be run as system jobs.

First, remember that when a Pascal–2 program is first executed, the initialization code performs at least these steps:

1. Test for proper compiler version number;
2. Allocation of stack and heap, using the monitor's **.SETTOP** request;
3. Initialization of file channel areas.

Under all monitors, **.SETTOP** specifies a new program high address from the monitor, and Pascal–2 uses this new space for the stack and the heap. The stack is used to store local variables and return addresses, and to pass parameters to procedures and functions. The stack starts at the address specified in the **.SETTOP** instruction and expands downward. The heap begins just after the program image and grows upward toward the stack. The heap is allocated dynamically with Pascal's predefined procedure **new**. Memory is returned to the heap when files are closed or when the predefined procedure **dispose** is used to deallocate variables allocated via **new**.

In the XM monitor, **.SETTOP** operates differently than it does in the SJ and FB monitors, in that enabling the special **.SETTOP** requires two actions: linking with the **/XM** switch and making the job virtual. Thus, **.SETTOP** allocates memory (hence the stack and the heap) from extended memory. Pascal–2 allocates this space with the instruction

```
.SETTOP #-2
```

With XM executing the virtual job, the Pascal program has 32K words of address space.

With the stack and heap in extended memory, it would be useful to put the whole Pascal program there also. All virtual programs require a static region that must reside in low memory, to hold the root and low memory overlays. However, in order to minimize the amount of low memory required, overlays aren't used and the root is made as small as possible. Instead, create a module with a new transfer address that will reside in low memory but only consists of a jump to the Pascal–2 entry point. This assembly-code module consists of:

```
        .GLOBL   $START
XMST::  JMP      $START
        .END
```

To link the program with the module, use the following instructions:

```
.LINK/XM/EXE:PROG/PROMPT/C
*/TRANSFER XMST,VIRJOB,PASCAL
*PROG/V:1//
Transfer address? XMST
```

This method puts much of the code into extended memory. Due to the way the Linker loads library code, the root still contains the initialization code and most of the other library support routines and its size remains 2600 words. For this technique to be successful, the reference to the library routines must be forced into the overlay region. Again, you use the assembly code routine:

```
.GLOBL    XMST
XMST1::   JMP      XMST
.END
```

and link the program with these commands:

```
.LINK/XM/EXE:PROG/PROMPT/C
*/TRANSFER XMST1,VIRJOB,PASCAL
*XMST,PROG/V:1//
Transfer address? XMST
```

This puts all but the overlay handler with its table and the **JMP XMST** instruction into virtual overlay region 1. The size of the root has been reduced to less than 400 words, as you can see using the **SHOW JOBS** command.

Creating Pascal system jobs in this manner carries a couple of restrictions:

- The addition of any more segments and/or regions will put Pascal–2 initializaton code and I/O support back into the root. The program then needs a minimum of 2400 words of low memory and must be linked differently.
- The allocation of 32K words of memory by the Pascal–2 support library **.SETTOP #-2** instruction restricts the number of jobs you can run. For example, on a 192K-byte system, memory can only hold three jobs (with the queue running) before there is insufficient memory.

- When the virtual high limit of these programs lies within 28K to 32K words of memory (page 7), the job fails during initialization at the call to .SETTOP. Although such cases are rare, these failures are caused by the manner in which .SETTOP allocates memory above the user program upon initialization.

Although DEC insists that RT is a single-user system, under certain circumstances we have found that it can almost look like a multi-user system. In this case, we have been able to run data entry applications and large compute-bound jobs that call .TWAIT, allowing background programs time to run also. We intend to use this approach for multiterminal applications in the near future.

Reserving room for RT-11's USR

Mr. Siemens is with Omnex Coporation, 801 East Charleston Rd., Palo Alto, CA. Omnex uses Pascal to develop system utilities and end-user applications. One such application is a local system network providing a disk server to a number of remote RT-11 systems.

This article describes NEWSTART, a small initialization routine that reserves enough space for the User Service Routine (USR) to remain resident in an RT-11 system with the single job (SJ) monitor. By eliminating the USR swapping that occurs when a file **reset** or **rewrite** is performed, this routine speeds up the file opening process.

Under RT-11, programs compiled with either Pascal–1 or Pascal–2 are normally linked with a starting address at global symbol $START. At this address, the initialization code allocates the stack to be used by the program. The default stack is placed at the top of memory (a .SETTOP for all of memory is performed) unless the user has specified a stack address with the linker options /STACK (CCL syntax) or /M (CSI syntax). Actually, these options simply load a value that indicates the initial setting of the stack pointer into location 42 of the program. If the initialization code at $START finds that location 42 is greater than the high program limit, then it will use the value in location 42. Otherwise, it will ask for all of memory.

The NEWSTART routine first tests to see which operating system is being used. If it is TSX-Plus or a non-privileged virtual job under the XM monitor, then control is immediately transferred to the regular start location, $START. If it is RT-11 SJ and SET USR SWAP is in effect (allowing the USR to swap), then location 42 is set with the normal USR load address. The routine at $START will then allocate the stack below the USR.

The initialization routine is simply linked with the program and a specific transfer address to global symbol NEWSTA specified. For example, assume the routine exists in the object file NEWSTR.OBJ, which contains global symbol NEWSTA where the program should start. The following commands link a program with NEWSTR:

```
LINK/TRANSFER PROG,NEWWSTR,SY:PASCAL
Transfer symbol? NEWSTA
```

As a demonstration, the following short Pascal–2 program opens an output file 10 times and measures the elapsed time required. The table below the program shows the measured execution times for this program with and without NEWSTR on three different operating systems.

```
program Stest;
  var
    I: integer;
    F: text;
    T1, T2: real;
  begin
    T1 := time;
    For I := 1 to 10 do
      begin
      rewrite(F, 't.tmp'); requires use of USR
      close(F);
      end;
    T2 := time;
    writeln('Elapsed time', 3600.0(T2 - T1): 6: 2, ' seconds');
  end.
```

Measured Execution Times in Seconds

Operating System	with NEWSTR	without NEWSTR
TSX-Plus	1.39	1.39
RT-11SJ	1.00	8.85
RT-11FB background	1.00	9.63

RT-11 USR Continued from Page 5

The source code for the new initialization routine follows. Note that it does not occupy any extra space in the save image file because it resides in psect RTSDAT (an overlaid psect). After NEWSTA has run, the code at $START loads data over it.

```
.TITLE   NewStart for pascal programs
.Enable  lc

;-----------------------------------------------------------------;
;                                                                 ;
; Module:      NewStart                                           ;
;                                                                 ;
; File:        NEWSTR.MAC  (Source)                               ;
;              NEWSTR.OBJ  (Object)                               ;
;                                                                 ;
; Last Chng:   18-Aug-82   9:16 AM                                ;
;                                                                 ;
; Purpose:     This module starts Pascal-1 and Pascal-2 programs  ;
;              in such a way as to leave room in high memory for  ;
;              the USR.  This will keep USR swapping from occurring ;
;              when files are opened without forcing the user to do ;
;              an explicit SET USR NOSWAP.                        ;
;                                                                 ;
;                                                                 ;
;-----------------------------------------------------------------;


; When linked such that the symbol NEWSTA is the transfer address,
; this routine makes sure that sufficient space is left in high memory
; for the USR to remain resident.  This space is reserved only
; if the USR is not already available.

        .PSECT  RTSDAT, RW, I, GBL, REL, OVR

; This psect contains data necessary for the RT-11 interface.
; It is an overlaid psect, and since it contains no valid data until
; initialized, we can use the space for this one time only
; initialization code.

        .GLOBL  $START,RtArea
        .MCall  .GVal, .Serr, .Herr

        FrstStk   =   42                    ;contains initial top of stack
        Jsw       =   44                    ;job status word
        VirJob    =   20000                 ;virtual job bit in jsw
        UsrLdAdd  =   266                   ;offset for USR load address
        Config    =   300                   ;offset for configuration word
        UsrLkd    =   1000                  ;USR locked indicator bit
```

```
NewStart::
        .Serr                           ;soft errors desired
        Mov     #TsxGln,R0
        Emt     375                     ;try to get TSX-Plus line number
        Bcc     1$                      ;it worked, don't need to reserve
                                        ;any room for USR with TSX

        Bit     #VirJob,@#Jsw           ;check for virtual job under XM
        Bne     1$                      ;it is, USR is available under XM

        Cmp     #1000,@#FrstStk         ;check for initial stack = 1000
        Bne     1$                      ;branch if it is
        .GVal   #RtArea,#Config
        Bit     #UsrLkd,R0              ;check for USR locked
        Bne     1$                      ;branch if it is
        .GVal   #RtArea,#UsrLdAdd       ;get USR load address
        Dec     R0
        Dec     R0
        Mov     R0,@#FrstStk            ;set initial stack
1$:     .Herr                           ;enable hard errors again
        Jmp     $START                  ;go to normal run-time initialization

TsxGln: .Byte   0,110                   ;code for get line number

        .End
```

Reducing a Pascal-2 program's task image size

Users who are running out of disk space (who isn't?) can use an obscure Task Builder option to reduce the size of their Pascal–2 task image files. The room saved depends on the size of the global data area in the program. Consider the following short program:

```
program Test;

  var I: integer;
    A: array [1..20,000] of integer;

  begin
    for I:=1 to 20,000 do
    A[I]:=I;
  end.
```

After this program is compiled, it can be linked in the usual manner, using the command:

```
>TKB TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
```

The size of the task image is 92 blocks. The image is so large because the array **A** allocates 20,000 words of storage in the global data area, represented by the program section called **GLOBAL**. However, Pascal does not initialize this data area. This means that the data area need not be present in the task image. To get rid of the allocation for the psect GLOBAL, create the following overlay description file (TEST.ODL).

```
.NAME PASDAT,NODSK
.PSECT GLOBAL,D,GBL,OVR
.ROOT TEST-LB:[1,1]PASLIB/LB,PASDAT-GLOBAL
.END
```

The **.NAME** directive gives a name to a segment (called **PASDAT** here) and assigns the **NODSK** attribute to that segment. This means that the Task Builder will not allocate disk space for this segment. The **.PSECT** directive defines the program section **GLOBAL** in which Pascal will allocate all of the global level variables. The psect is named here so we can force it into the **PASDAT** segment. This is done by creating a co-tree in the **.ROOT** directive. The effect is that no disk space will be allocated for the co-tree called **PASDAT**. This causes no problems for the program, because all of the data in the section **GLOBAL** is initialized at run-time.

Use the **/MP** switch to tell the Task Builder to use the overlay description in the file TEST.ODL and build the task as shown below:

```
>TKB TEST/FP/CP,TEST=TEST/MP
```

Now the size of the task image file is 14 blocks. This is a considerable savings in disk space! This technique does not save any memory. In fact, the task grew by about 100 words by the addition of the co-tree. This trick works because the Pascal data area does not need to be read in from the disk to be initialized. The task loader allocates the space automatically when the task is activated.

Using literal strings in structured constants

As some users have noticed, Pascal–2's output files contain duplicate entries of literal strings that are contained within structured constants. This is due to the compiler's multi-pass design. On the first pass, all strings are placed in the output file. During the second pass the structured constant is processed and the entire constant value is placed in the output file, including a copy of any literal strings.

Example 1: Regular Use of Structured Constants

Input code in program:

```
type
 Rec = record
  Txt: packed array [1..12] of char;
  Len: 0..12;
  end;
 {$nostandard}
const
 Duplication = Rec('duplicate   ', 9);
 {$standard}
begin
end.
```

Output code in the CONST psect:

```
.psect consts,d,con,lcl
$const:
 .word 72544 ;'ud'  --literal string
 .word 66160 ;'lp'
 .word 61551 ;'ci'
 .word 72141 ;'ta'
 .word 20145 ;' e'
 .word 20040 ;      '  '
 .word 72544 ;txt = 'ud' --structured const
 .word 66160 ;      'lp'
 .word 61551 ;      'ci'
 .word 72141 ;      'ta'
 .word 20145 ;      ' e'
 .word 20040 ;      '  '
 .word 11    ;len = 9
```

Users who require absolutely minimal-size programs, e.g. for ROM applications, may use this simple work-around. Instead of defining the strings as strings, drop down one level of structuring and define the individual characters of the array as separate elements. This method requires longer input code but generates only one copy of literal strings, producing shorter executable code in the **CONSTS** psect, as the following two examples show.

Example 2: The Work-around

Input code in program:

```
type
 Rec = record
  Txt: packed array [1..12] of char;
  Len: 0..12;
  end;
 {$nostandard}
const
 NOduplication = Rec(('n', 'o', 'd', 'u',
                      'p', 'l', 'i', 'c',
                      'a', 't', 'e', ' '),
                     11);
 {Note the additional parentheses needed
  to indicate a lower level of structuring.}
 begin
 end.
```

Output code in the CONSTS psect:

```
.psect consts,d,con,lcl
$const:
 .word 67556 ;'on' --structured const
 .word 72544 ;'ud'
 .word 66160 ;'lp'
 .word 61551 ;'ci'
 .word 72141 ;'ta'
 .word 20145 ;' e'
 .word 13    ;len = 11
```

The Log: Pascal-1 and Pascal-2

The Log is empty for this issue. We are still releasing version K of both Pascal–1 and Pascal–2 as our current product. If you have any problems with version J or before, and you are still in support, be sure to request your free update. We have not forgotten those of you who have submitted Trouble Reports for Version K. At this very moment,we are busily fixing bugs for Pascal–1 Version 1.3 and Pascal–2 Version 2.1.

In addition, we are adding a number of new features to Version 2.1, such as an improved interface to the Pascal support library, improved error reporting, and conformant arrays.

When the Support Group at Oregon Software gets several calls about a particular problem, we do one of two things. We fix bugs, announcing the fixes in the Log, and we publish articles in the newsletter, to clarify the use of Pascal–1 or Pascal–2 under the various operating systems, or to offer work-arounds for specific users' problems. Your letters help us to be aware of current problems facing users, to provide specialized information on the operating systems we support, and to develop the support libraries for our products. So, if you don't find information you need in the newsletter, write us.

Editor's note: The yearly support fee for Pascal-2 is $900; Pascal–1 support is $600 annually.

Bug contest begins

Why let program bugs bring you nothing but grief? Enter Oregon Software's Bug Contest and win a prize for the most interesting bug you've created while writing programs using our software.

What do I enter? You can enter the contest in any or all of three categories: most interesting bug, most interesting application program, most interesting prize.

We all know what a bug is. A mistake or oversight in program logic can be subtle and conditional, yielding unanticipated or undesired results. Such bugs are intellectually interesting as puzzles. What makes one of them more interesting than another? The bug really captivates us when it causes truly spectacular results, like turning all the traffic lights in the city red for an hour.

An interesting application could be an innovative use of Pascal, some heretofore unrecognized characteristic of the language finding its place in an area of the real world where Pascal is not usually found. Or, a program could be interesting because of its unusual results, the job it accomplishes. Or, the code of the program could be interesting in its own right, because the algorithm or its coding is so clever and elegant. We'll accept all three types.

Finally, the prize suggestions will be interesting because of the thought, not the expense. The best birthday gift is "right" because it fits the person better than anything that person would have asked for. We look for suggestions that are exquisite, appropriate, and attainable. Because the winner of each category gets one of the top three prizes, we hope your suggestions are not so unique that we can't obtain them.

How do I enter? You'll notice that we've set up the entry blank in three parts: the top two for events and applications, and the bottom for prize suggestions. You may enter all three, and you may submit as many entries as you like. Make as many photocopies of the entry form as you need.

In the description of your bug and/or application, please include these bits of information:

- We'd like to know the environment of your specimen. What conditions caused the bug to break out of its cocoon and take flight?
- Where and when did you develop the application program? What problem did the program solve?
- Were there any unusual aspects of your hunt for the bug or your development of the application program?

Other bug catchers will be curious about the handling of such pests, so please tell us what you did to squash your bug, and what you think others could do to avoid such occurrences.

And if I win? For those of you who don't think the fun of entering is its own reward, we've that third category for suggested incentives. We plan to give the three most interesting prizes to each winner. The keeper of the most interesting bug will be awarded the most interesting prize suggestion. The second place prize suggestion will go to the submittor of the winning application program. The person who suggests that most interesting prize gets the third most interesting prize (after all, you don't want to know what you're getting, do you?).

When is my entry due? All entrants and the winners will be announced in Newsletter #7 (June 1983). So, we need to have your entry by March 1, 1983. We already have several entries, so don't delay. Send yours now!

Pascal NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

Collins Hemingway, editor
David Spencer and Michael Kuhn, staff writers

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. Fee: $100 per newsletter page. Payment upon publication.

Printed in USA

ENTRY BLANK

Name _____ **Site #** _____

Company Name _____

Description of Entry:

BUG

APPLICATION

PRIZE

CONSUMER SURVEY

This survey of Oregon Software's customers and other interested parties will provide information for us to use in planning over the next year. We look forward to receiving your answers promptly. To encourage participation, we are offering a $100 discount off any of our products or services to everyone who returns a completed survey. To receive the $100 coupon, fill in your name and address on the final survey question. To use the coupon, simply include it with any order and subtract the $100 from the amount you owe us. The coupon expires May 30, 1983.

After completing the questionnaire, please follow the mailing directions below.

Mailing Directions

`Domestic`: Tear off the questionnaire section and fold it so that the BUSINESS REPLY MAIL permit faces out. Secure the open edge with staple or tape, then mail.

`International`: Please enclose the questionnaire pages in an envelope and return them to your distributor. If you do not have a distributor, return the questionnaire directly to us. (Unfortunately, we cannot supply prepaid postage for overseas respondents.)

Thank you in advance for your response.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. A407 PORTLAND,OR.

POSTAGE WILL BE PAID BY ADDRESSEE

OREGON SOFTWARE INC.
Attn: Marketing Survey
2340 SW Canyon Road
Portland OR 97201

1. What Oregon Software products does your organization currently have?

___ Pascal-1
___ Pascal-2
___ Sort-1 or Sort-1-Plus
___ None

2. On what processor/operating system is your Pascal compiler hosted?

___ PDP-11/RSX ___ PDP-11/RSTS
___ VAX/VMS ___ PDP-11/RT-11 or TSX
___ PDP-11/RSX ___ Does Not Apply

3. How important are the following Pascal compiler features to you?

		Unimportant			Important		
a.	Code Size	1	2	3	4	5	N/A
b.	Code Execution Speed	1	2	3	4	5	N/A
c.	Compiler Execution Speed	1	2	3	4	5	N/A
d.	Portability of Code	1	2	3	4	5	N/A
e.	Adherence to Pascal Std.	1	2	3	4	5	N/A
f.	Good Support	1	2	3	4	5	N/A
g.	Ease of Debugging	1	2	3	4	5	N/A
h.	Utilities Package	1	2	3	4	5	N/A
i.	Competitive Price	1	2	3	4	5	N/A
j.	Availability of Extensions	1	2	3	4	5	N/A
k.	Other (please specify)_____	1	2	3	4	5	N/A

4. Do you use Oregon Software's support services?

___ Yes ___ No ___ Does Not Apply

If yes, how satisfied have you been with the support?

Very Dissatisfied Very Satisfied
1 2 3 4 5

If dissatisfied, or if you are no longer in support, explain why.

_____

_____

_____

5. How would you compare the cost of completing a programming project using Oregon Software's Pascal versus another approach?

   a. ___Far Below ___Below ___About the Same ___Above
      ___Far Above ___Unable to Determine

   b. What would the other approach be?_____

6. Does Oregon Software's Pascal-2 have any serious limitations or drawbacks?

   ___Yes ___No ___Does Not Apply

   Please explain (use extra sheet if necessary)_____

   _____

   _____

7. Please indicate the three most important reasons for choosing Oregon Software Pascal over another compiler or language.

   1. _____(First Priority)

   2. _____(Second Priority)

   3. _____(Third Priority)

8. What is the job title and function of the person most influential in deciding to acquire a compiler?

   _____

   _____

9. a. What is the basic activity at your facility?

   ___ Components/sub-assemblies
   ___ Computers/peripheral equipment
   ___ Communications systems/equipment
   ___ Aircraft/space/ground support equipment
   ___ Test/measurement/instrumentation/process control equipment
   ___ Other (please specify)_____

   b. What is the basic function performed at your facility?

   ___ Research and development
      ___ Hardware ___ Software
   ___ Manufacturing
   ___ Service and support
   ___ Other (please specify)_____

c. Which primary classification applies to your facility?

___ Commercial
___ Government agency (non-military)
___ Military
___ Education
___ Other (please specify)_____

10. What is the approximate size of the programming staff at your facility?

a. ___ Under 5 ___ 11 to 20 ___ 30 to 50
___ 5 to 10 ___ 21 to 30 ___ More than 50

b. How many of these programmers use Pascal regularly? _____

11. Please indicate how important the following languages are to you.

		Unimportant			Important		
a.	FORTRAN	1	2	3	4	5	N/A
b.	BASIC	1	2	3	4	5	N/A
c.	Assembler	1	2	3	4	5	N/A
d.	Pascal	1	2	3	4	5	N/A
e.	COBOL	1	2	3	4	5	N/A
f.	C	1	2	3	4	5	N/A
g.	ADA	1	2	3	4	5	N/A
h.	PL/1	1	2	3	4	5	N/A
i.	Other (specify) _____	1	2	3	4	5	N/A

12. What is your job title and function?_____

Did you have major influence on the decision to buy a compiler?

____ Yes ____ No

13. Where did you first hear about Oregon Software?

___ Friend or associate ___ Magazine article
___ Trade show ___ Magazine ad
(which magazine?_____)
___ Professional association ___ In school
___ Distributor ___ Other (specify)_____

14. How important are the following sources for information on software?

	Unimportant			Important		
a. Vendor publications	1	2	3	4	5	N/A
b. Distributors	1	2	3	4	5	N/A
c. Sales representatives	1	2	3	4	5	N/A
d. Magazines	1	2	3	4	5	N/A
e. Trade shows	1	2	3	4	5	N/A
f. Technical seminars	1	2	3	4	5	N/A
g. Friends and associates	1	2	3	4	5	N/A
h. Other (specify) _____	1	2	3	4	5	N/A

15. What job-related publications do you read regularly?

_____ _____

_____ _____

_____ _____

16. To receive coupon:

Name _____

Organization _____

Street Address _____

_____

City, State _____

Country, Postal Code _____

Pascal NEWSLETTER
OREGON SOFTWARE
2340 SW Canyon Road
Portland, Oregon 97201

Pascal
NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

Pascal
NEWSLETTER

NUMBER 6 OREGON SOFTWARE SPRING 1983

Pascal-2 Version 2.1 to be released

Oregon Software now has Version 2.1 of our optimizing Pascal compiler in field test for RSX, RSTS, and RT-11 systems. Version 2.1 for all systems will be released to general users June 1, beginning with RSX.

The upgrade from V2.0 represents a major revision of the Pascal–2 software, to include conformant array parameters, improve error diagnostics, enhance performance, remove size limitations on user programs, and fix all major bugs collected in the last year.

Immediately after field test, Version 2.1 will be released to all customers who have reported bugs since Version 2.0K was released last year. These shipments will probably be concluded before June 1. Then, the V2.1 update will be released to all supported customers who send us a written request.

With the energy that has gone into producing V2.1, we did not release any interim updates after V2.0K. For this reason, we will send Version 2.1A to any customer who renewed support after the release of V2.0K, even if that customer's support expires before the release of V2.1A. Updates are free on floppies and magtape; a fee is charged for other media. All customers must request V2.1 in writing.

Because many new features have been added and because the code generated by the V2.1 compiler is different from that of V2.0, users will need to modify and recompile existing Pascal–2 programs once they have installed V2.1. Users may wish to redesign existing programs to take advantage of the new features such as conformant array parameters, lazy I/O and user control of run-time error reporting.

With the addition of conformant array parameters, the Pascal–2 compiler conforms to Level 1 of the proposed ISO standard (ISO dp7185). With conformant array parameters, users can write general procedures that accept arrays with different lower and upper bounds. With V2.0, users have to write a new procedure for each array that varies in size or bounds.

We have substantially improved the error diagnostics by including a procedure walkback generated as a result of a run-time error. The walkback displays the error message, the point of error in Pascal source terms, and a traceback of procedure calls leading to the error. Version V2.0 has no equivalent run-time diagnostics.

In addition, a new I/O error trapping capability allows users to recover from I/O errors with their own code, to change the wording of run-time error messages and to print additional information besides that printed in the walkback.

Other substantive changes include: the addition of "lazy I/O," a scheme by which data is not read or written until actually used in the program; elimination of certain size restrictions, allowing larger programs to be compiled; capability to include more procedures and type definitions; new procedures to do common and useful I/O procedures, including delete and rename; and features to help users of small systems.

Changes and additions to V2.1 are documented in a 70-page supplement to the *Pascal–2 User Manual*. The supplement will be supplied with all updates. The changes will be included in a new edition of the manual to be printed in the late spring.

This newsletter carries a summary of the major changes, plus articles on the use of various new features. Major bug fixes are described in the Bug Log.

In this issue...

Pascal–2 Version 2.1 to be released . . . . . Page 1
UNIX version joins Pascal-2 family . . . . . Page 2
SourceTools available for RSX . . . . . Page 2
Summary of changes/new features for V2.1 . Page 3
Terminal I/O . . . . . Page 4
Single-character I/O . . . . . Page 4
FORTRAN carriage control . . . . . Page 5
I/O error trapping . . . . . Page 6
Random access to text files . . . . . Page 8
Lazy I/O . . . . . Page 10
Converting unsigned integers . . . . . Page 11
The Log: Pascal-1 and Pascal-2 . . . . . Page 12
Information exchange . . . . . Page 13
OPUS Communiqué . . . . . Page 14
Marketing group evolves . . . . . Page 15
User manuals for V2.1/1.3 . . . . . Page 16

UNIX version joins Pascal-2 family

Oregon Software has released the Pascal-2 compiler on the UNIX operating system for the PDP-11. The new product includes an enhanced version of our interactive source-level debugger.

UNIX, originally developed by Bell Laboratories, is sold through license agreements with Western Electric. Popular in university environments originally, UNIX is becoming a major force in software development applications and has spawned many UNIX "look-alike" operating systems.

A key feature of UNIX is the collection of supporting "tools" that provide a software developer with standard, easy-to-use programs to help with coding and development. Oregon Software's Pascal compiler system contains similar kinds of tools — debugger, execution profiler, and development utilities.

The debugger allows the programmer to work interactively at the source level to solve logic errors in applications, thus simplifying and speeding development. The debugger runs as a separate process from the application code that is being debugged, keeps track of multiple compilation units, and performs breakpoint debugging without slowing down the application code.

The profiler aids the applications programmer by identifying execution bottlenecks. With data supplied by the profiler, the developer can reorganize portions of the code to substantially improve overall execution speed.

The Oregon Software Pascal-2 UNIX compiler supports all capabilities of standard Pascal and conforms to the draft proposed Pascal standard (International Standards Organization dp7185.1, level 0). As a result, users are assured that code developed under Pascal-2 can be moved to other systems with standard Pascal compilers.

Pascal-2 for UNIX is available now directly from Oregon Software or from authorized distributors.

SourceTools available for RSX

Oregon Software is now releasing SourceTools for RSX, including VMS in compatibility mode, to general end users. The seven-program package helps programmers develop and coordinate large software projects. SourceTools has been in field test for the last three months and no major problems have been reported.

The source-control system, SourceCon, is the foundation of the package. SourceCon monitors files placed under its control, to prevent programmers from making simultaneous and conflicting changes and to keep a development history of the source files. In a sense, SourceCon provides a "library service" for programmers and writers, recording changes to text files and the reasons for them, and ensuring that earlier versions of a module are always available.

Another program in the SourceTools package simplifies and automates the task of rebuilding software from component parts. The SourceTools MAKE program automatically determines which files are out of date, then executes only the commands needed to rebuild those out of date files.

In a more conventional environment, the programmer must remember how all the components of a large program fit together whenever some components are modified, and must recall exactly how to re-create each component so that it fits with others in the module. This is the old "recompile everything in sight" problem. Or, the programmer has to keep a set of command files around for each of the various parts and for combining the parts together.

Two additional utilities alleviate the tedium of maintaining parallel or alternate versions of files. A text comparison program, TEXTCOM, isolates differences between two text files. A stream editor, SEDIT, interprets a text-editor script and applies the editor-commands to its input text file. By a switch in the text comparison program, TEXTCOM can output an editor script compatible with SEDIT. This combination can be a powerful tool for maintaining parallel files on remote systems and for cross-development work.

These time-saving software tools are used extensively by our own programmers.

Summary of changes and new features for Pascal-2 Version 2.1

New features provide the following benefits for users:

- Conformant array parameters allow you to write general procedures that accept array parameters of different sizes and with different lower and upper bounds.
- Non-decimal constants allow you to use constants in number bases ranging from base 2 (binary) to base 16 (hexadecimal).
- Run-time error walkback aids in diagnosis of run-time errors.
- User-processing of run-time errors gives you control of run-time error reporting. You can change the wording of run-time error messages and print additional information besides the walkback. The walkback may be disabled with the **nowalkback** switch.
- I/O error trapping capability allows you to recover from I/O errors with your own code. Article on page **6**.
- A new procedure provides an explanation of I/O errors (RSTS and RSX only). Article on page **6**.
- Lazy I/O. A new I/O scheme in which data is not read or written until actually used in the program. Article on page **10**.
- New I/O control switches extend the capabilities of the file system. See related article on page **10**.
- New procedures **getpos** and **setpos** simulate random access to files of type **text**. Article on page **8**.
- New syntax of **%include** directive allows you to specify the disk volume number, UIC and version number of the included file.
- New built-in procedures **rename** and **delete** allow you to rename or delete files from within the program.
- New function **space** determines the amount of stack and heap available to an executing program.
- Eight-bit characters allow you to use extended character sets.
- **WK**: specification allows you to specify the device to which the compiler directs its working data files and helps small-systems users reduce load on the default disk.

The following changes have been made:

- Certain size restrictions have been eliminated to allow larger programs to be compiled. Also, V2.1 programs can have more procedures and more type definitions than V2.0 programs.
- New run-time error numbers and messages have been added.
- Single-character I/O has been added to RSX. Article on page **4**.
- New method of reading MCR command lines has been added to RSX.
- New ways to overlay Pascal–2 programs; new way to overlay the Pascal–2 support library.
- New support library interface. Library entry points have been changed to the form **P$***nnn*, where *nnn* is a number in the range 0..135.
- Procedure **timestamp** added to RSX. Returns the date and time.

Pascal-1 V1.3 in field test

Version 1.3 of Pascal–1 is also in field-test for RSX, RSTS, and RT-11 systems. V1.3 also will be made generally available to end users by June, along with Pascal–2 V2.1.

V1.3 will include lazy I/O, a new library interface, and a new initialization interface. Pascal–1 users will be able to use the new I/O error diagnostics, but they must change their source programs.

A number of bugs also have been fixed.

Terminal I/O for RSX

For each operating system, the terminal driver acts as an intermediary between a Pascal program and the terminal that is running that program. The terminal driver is record-oriented, which means it sends a full line of text to the screen or to the program instead of sending one character at a time.

For example, when a **read** statement reads input from a terminal, the terminal driver reads each character and places it in an internal buffer until the terminal driver encounters an end of line (a carriage return). At this point, the terminal driver sends the entire line to the program and lets the program process the line one character at a time.

The same is true for **write** statements, except that the terminal driver buffers the characters being sent to the terminal until a **writeln** clears the buffer or until the buffer size is reached. Then the record is displayed on the screen. The size of the buffer can be controlled with the **buffersize**:*n* file control switch. See "Single-Character I/O for RSX."

As mentioned above, the **writeln** statement instructs the terminal driver to send the output to the screen. After the terminal driver prints a line, it positions the cursor over the first character of the line it just printed. The usual output sequence sent by the operating system's terminal driver is: line feed, data, carriage return. This sequence moves the cursor to the next line, prints the data and returns the cursor to the first position of the newly printed line. This method of writing lines to a terminal is different from other operating systems where the normal output sequence is data, carriage return, and line feed (print the record, move to the first character of the line, then move to the next line).

The normal sequence of commands issued by the terminal driver may not particularly suit special I/O applications such as direct cursor addressing. On RSX, for example, the line-feed and carriage-return characters are often not needed or are sometimes unwanted. To gain control of the terminal-I/O command sequence, use the **ftn** file control switch , a feature which allows you to override the effects of the terminal driver (see "Using FORTRAN Carriage Control under Version 2.1 for RSX").

Single-character I/O added for RSX

The **buffersize**:*n* I/O control switch sets the line size of a **text** file to *n* bytes. By setting your terminal's internal line size to 1 with **buffersize**:1, you can write programs that perform single-character input and output. This feature is useful when you need to do special output formatting to a video terminal, such as direct cursor addressing of output on the screen.

CAUTION

Single-character input/output increases system overhead and should be used with care. Overhead is increased because the monitor must be called for each character read from and written to the terminal. Several users simultaneously outputting in single-character mode can significantly reduce the system response time.

To enter single-character mode, specify the **buffersize**:1 switch on the **reset** statement for single-character input and on the **rewrite** statement for single-character output. In single-character mode, the terminal driver reads and writes the characters as usual. But since the internal buffer size is one byte, each new character fills the buffer, causing the buffer to be emptied. In other words, each character written via **write** is immediately printed on the screen; each character that you enter is immediately sent to the program.

Continued on Page 8

```
program Single;
  var
    Ch: char;

  begin
    reset(input, 'TI:/buff:1');
    rewrite(output, 'TI:/buff:1');
    write('Type a message: ');
    while not eoln do
      begin
        read(Ch);
        write(Ch);
      end;
  end.

>RUN SINGLE
Type a message: ddooùubbllee_ yyiissiioonn
>
```

Program Single uses a combination of single-character input and single-character output procedures under the RSX operating system. When compiled and linked and run, the program simply echoes each entered character as shown.

Using FORTRAN carriage control under Version 2.1 for RSX

Pascal–2 allows your Pascal programs to write **text** files that follow the FORTRAN standard output conventions. To do this, specify the **ftn** file control switch on the **reset** or **rewrite** statement that opens the file. For interactive I/O, the **ftn** switch is used with the standard file **output**.

FORTRAN conventions state that the first character of each line of an ASCII file is nonprintable and is used to control the vertical formatting of an output file or terminal screen (see table of commonly used vertical formatting characters). Most characters have no useful meaning in the first position; however, some characters significantly affect the format of the output. The *RSX-11M/M-Plus I/O Drivers Reference Manual* contains a complete list of these vertical formatting characters.

The null character (**chr(0)**) can be used to prevent the terminal driver from adding any special characters to the data you write to the terminal. You will have to insert the carriage-control characters yourself, as shown in the last line of sample program **Ftn**. In the output shown for program **Ftn**, the "overprint" **writeln** replaces the characters "Normal output" from the previous **writeln** with "Overprint****," resulting in a different line of text. If you set your terminal to a slow baud rate, you would actually see the overprinting occur. If you direct the file to a printer, the overprinted line will contain overstrikes.

The **ftn** switch does not solve all terminal I/O problems. The data is not written to the terminal immediately; the characters are still buffered until a **writeln** is executed. You can overcome this problem by using the feature that allows single-character I/O.

```
program Ftn;

  begin
    rewrite(output, 'TI:/ftn');
    writeln(' Normal output');
    writeln(' More normal output');
    writeln('0Double space');
    writeln('1Page eject');
    writeln(' Normal output of a long line');
    writeln('+Overprint****');
    writeln('$Prompt output: ');
    writeln(chr(0), 'Internal format',
            chr(13), chr(10));
  end.


>RUN FTN
Normal output
More normal output

Double space

<ff> ———————— skips to top of next page

Page eject
Overprint**** of a long line
Prompt output: Internal format

>
```

Program Ftn shows the use of FORTRAN standard output conventions in a Pascal program under RSX. The first character written on each line is interpreted as the vertical-format control character. **Chr(13)** is the carriage-return character, and **chr(10)** is the line-feed character. The output shown appears on your terminal when the program is run.

Commonly Used Vertical Formatting Characters

Character	Meaning	Output Sequence
<space>	Normal output	Line feed, data, carriage return
0	Double-space	Two line feeds, data, carriage return
1	Page eject	Form feed, data, carriage return
+	Overprint	Data, carriage return
$	Prompt	Line feed, data, remain on same line
<null>	No special formatting	Data only

New routines trap I/O errors

Pascal-2 V2.1 permits you to write programs that trap and detect many kinds of I/O-related (normally fatal) errors. Using three predefined routines to process I/O errors with your own code, you can terminate the program at the occurrence of an I/O error or continue execution in spite of the error. You can print your own diagnostics with a fourth procedure.

The three error-trapping routines — procedure **noioerror** and functions **ioerror** and **iostatus** — are predefined and do not need to be declared in your program. They accept a file variable as their only parameter. The **sayerr** procedure, which prints the text of a given error messge, is not predeclared, so you must declare it when used. Although details vary slightly according to the operating system, these routines routines work as follows.

Procedure **noioerror** specifies that the calling program will handle any I/O errors that result from reading or writing to the specified file. The file must be open before **noioerror** is called.

Function **ioerror** determines the status of the last I/O operation that the program performed on the specified file. This **boolean** function returns a **true** value if an I/O error has occurred or a **false** value if the operation was successful.

Function **iostatus** helps your program determine the cause of the error by returning an integer error code describing the last attempt to access a file. Your program can either bypass the problem and continue processing, or terminate so you can correct the problem.

You can pass **iostatus** as a parameter to **sayerr**, an **external** procedure which prints the text of the error message corresponding to the value returned by **iostatus**.

Continued on Page 7

Printing I/O errors under RSX

Procedure **sayerr** prints the text of a given error message. A negative error code indicates that the error is specific to the RSX operating system. A positive error code indicates that the error is detected by the Pascal-2 support library. Pascal-2 error codes, along with the text of the error message and a brief explanation of the cause, are listed in the V2.1 supplement to the *Pascal-2 User Manual*. RSX I/O error codes are listed in the *RSX I/O Operations Reference Manual*.

As an external procedure supplied in the Pascal support library, you must declare the **sayerr** procedure in your program as shown:

```
procedure Sayerr(Status: integer);
  external;
```

where **Status** is the error code of the error.

Procedure **sayerr** takes an RSX I/O error code (a negative number) and looks in the file LB:[1,2]QIOSYM.MSG to print the text of the error message. **Sayerr** only prints messages for negative I/O error codes in the range −255.. − 1; **sayerr** ignores error codes that lie outside this range, printing nothing.

```
program Opnerr;
  var
    F: text;
    Status: integer;

  procedure SayErr(Code: integer);
    external;
  begin
    reset(F, 'XXXX:', 'test.dat', Status);
    if Ioerror(F) then
      begin
        writeln('I/O status=', Iostatus(F));
        SayErr(Iostatus(F));
      end;
  end.
```

```
>RUN OPNERR
I/O status=    -55
Bad device name ———— message sayerr prints
>
```

Program Opnerr shows the use of procedure **sayerr** on RSX. The program attempts to open a file called TEST.DAT on a ficticious device with the name XXXX:. Normally, this program would fail with no specific indication of what caused the **reset** failure. The error is detected by **Ioerror**. Function **iostatus** returns the value -55 and passes the parameter to **sayerr**. The call to **sayerr** prints the text of the message for RSX I/O error code -55: "bad device name." When this program is compiled and executed on RSX, it yields output similar to that shown above. A RSTS example produces similar results.

```
program Iotest;

  var
    I,Times: integer;

  begin
    Noioerror(input);
    for Times := 1 to 5 do
      begin
        write('Type an integer: ');
        read(I);
        if Ioerror(input)
          then writeln('Error detected.  Status=', Iostatus(input))
        else writeln('The integer was: ', I: 1);
        readln;
        writeln;
      end;
  end.
```

```
>RUN IOTEST
Type an integer: 1234
The integer was: 1234

Type an integer: 123456789   ———————————————— integer too large
Error detected.  Status=    19   ———————— Pascal-2 error code for invalid integer

Type an integer: ffg   ———————————————— non-integer characters
Error detected.  Status=    19   ———————— Pascal-2 error code for invalid integer

Type an integer: 77
The integer was: 77

>
```

Program Iotest illustrates the use of the use of pre-defined error-trapping routines for RSX. This program continues to execute after an I/O error is found; without these routines, the first error would cause the program to abort. The call to **noioerror** notifies the run-time system that the program will handle errors detected on the standard file **input**. When compiled and run, the results of such a program are similar to the above. The first entry results in a successful read of the integer I. The second and third entries result in a Pascal-2 run-time error. The final entry is successfully read, and the program ends. RSTS and RT examples produce similar results.

When you call these routines, you are responsible for checking the status of each I/O operation, to ensure that it was successful. If you fail to check the status afterwards, the results will be unpredictable.

The I/O error-trapping procedures can be used to determine the reason that a file could not be opened. To use this feature, specify the fourth parameter on calls to **reset** and **rewrite**. The use of the fourth parameter keeps the **reset** or **rewrite** from trapping a normally fatal "open" error. This allows your program to recover and continue, or terminate under your control. (Sample program OPNERR demonstrates use of this feature under RSX.)

Random access to text files simulated

Because lines of text vary in length, the **seek** procedure cannot predict the location of a line within a text file. The Pascal-2 support library for V2.1 supplies two external procedures, **getpos** and **setpos**, that simulate random access to **text** files. Working together, these procedures use a set of values to locate the next line of text. **Getpos** determines the starting location of the line and **setpos** returns the file pointer to the specified starting location of a line within the file.

For text files, the beginning of each line is denoted by a block number and a byte offset into that block. Each block contains 512 bytes. The first line of a file starts with block 1, offset 0. If you try to access a nonexistent position or a position in the middle of a line, an I/O error will result. When the file is read, use **getpos** to determine the starting position of the next line, and save that block and offset combination for later use by **setpos**. The block number and byte offset must be values returned by **getpos**. You cannot compute the values yourself.

These two procedures are not predefined and must be declared in your program as **external**. These procedures do not provide "true" random access; you cannot use them to access a file you have not previously read.

Sample program REVERSE shows the use of **getpos** and **setpos**.

'Getpos' procedure

Procedure **getpos** determines the starting position of the next line to be read from or written to a text file. **Getpos** requires three parameters, which are passed by reference, as shown below. Together, the parameters **Block** and **Offset** point to the next line to be processed.

```
procedure GetPos(var F: text;
    var Block, Offset: integer);
    external;
```

where

F	is the file variable of type **text**.
Block	is the returned disk block number of the next line in file **F** to be read or written.
Offset	is the returned byte offset into **Block**.

You should always call **getpos** to obtain the location in the file before you call **setpos**, so the block and offset values being passed to **setpos** are valid.

'Setpos' procedure

Procedure **setpos** positions the file pointer to a specified block number and byte offset into that block. **Setpos** accepts the same three parameters as **getpos**, except **Block** and **Offset** are passed by value. The **setpos** declaration is:

```
procedure SetPos(var F: text;
    Block, Offset: integer);
    external;
```

where

F	is the file variable of type **text**.
Block	is the block number to which the file pointer is set.
Offset	is the byte offset into **Block**.

Together, **Block** and **Offset** point to the new position. To stress an earlier point, the block number and byte offset must be values returned by **getpos**. Do not attempt to compute the values yourself. Save the returned values for later use.

Errors

If an error is detected while **setpos** tries to position the file, the end-of-file flag **eof** is set to true. The **ioerror** and **iostatus** support library procedures may help you to determine the reason that the line could not be accessed. (For details on **ioerror** and **iostatus** see article on page _ .) If a file is positioned to a block and offset that does not correspond to the first character of a line, the results will be unpredictable.

Using FORTRAN... Continued from Page 4

No special formatting characters are inserted when you use the **write** statement (see "Using FORTRAN Carriage Control under Version 2.1 for RSX"). A **writeln** statement will print a carriage return followed by a line feed, as if the buffer were empty. You can supply formatting characters by using the **chr** function to generate the appropriate characters.

For single-character input, you do not need to type a carriage return after each character to signify the end of the line.

```
program Reverse;

{ Print the contents of a file in reverse line order }
  type
    Pointer = ^position;
    position =
      record
        Next: pointer;
        Block: integer;
        Offset: integer;
      end;
  var
    F: text;
    Filename: packed array [1..80] of char;
    P, X: pointer;

  procedure GetPos(var F: text; var Block, Offset: integer);
    external;

  procedure SetPos(var F: text; Block, Offset: integer);
    external;
  begin
    write('File name? ');
    readln(Filename);
    reset(F, Filename);
    P := nil;
    repeat                          { read the file }
      new(X);
      with X^ do GetPos(F, Block, Offset);  ———————————— get start of next line
      X^.Next := P;
      P := X;
      readln(F);
    until eof(F);

    while P <> nil do               { write the file }
      with P^ do
        begin
          SetPos(F, Block, Offset);  ———————————— set pointer to start of next line
          if not eof(F) then
            begin
              while not eoln(F) do
                begin
                  write(F^);
                  get(F);
                end;
              writeln;
            end;
          P := P^.Next;
        end;
  end.
```

Program Reverse demonstrates the use of the **getpos** and **setpos** procedures to simulate random access of text files on RSX, RSTS, and RT. The program reads a text file and saves the position of each line in a linked list. It then prints the file in reverse line order so that the last line of the file is printed first, the next-to-last line is printed second, and so on, with the first line printed last.

Lazy I/O interface for all systems

For standard Pascal, an interactive input file, such as a terminal, poses a problem. A program must always be able to determine the current status of an open file, i.e., it must be able to retrieve the current record from the buffer variable (F^) and to check the current values of **eoln** and **eof**. Since an interactive file is being created as it is being read, the program must periodically wait for additional input to determine the file's current status. With the Pascal support library's new approach to interactive I/O, the program should not wait for input at inconvenient times.

Pascal–2 Version 2.1 uses an input interface known as "lazy I/O" to handle input from **text** files. Since a file's status needs to be defined only when the program actually refers to it, lazy I/O safely delays any input operation until the program uses its results. When a Pascal program requests an input operation on a text file, the operation is recorded for later use. The delayed operation is triggered by any subsequent reference to the file's buffer variable, **eof** value, or **eoln** value. The delay is invisible to the program but is visible to the user from the way the program is synchronized with interactive input.

To use lazy I/O, you need to be aware of its effect on synchronization of input and output operations. As an example, consider a simple program that reads its standard file **input**, which is connected to a terminal. The program prompts for each line and stops at the end of the file. The design of the program is dictated by two requirements:

- For the prompt to be effective, it must appear before the user is required to type the line.
- To detect the end of the file correctly, the program must check for it before reading each line.

To meet both of these requirements, the program must print the prompt before performing any operation that requires the next line of the file to be known: e.g., checking for "end of file" or reading the line. The sample programs in the "Conversion Note" show the differences in design between Version 2.0 and 2.1.

CONVERSION NOTE

Lazy I/O changes the compiler's implementation of interactive I/O. Programs compiled with Pascal–2 Version 2.0 may have to be revised to conform with the new schema. For example, under V2.0 program INTERACTIVE is coded as follows.

```
program Interactive(input, output);
{sample for version 2.0}

  begin
    while not eof do
    begin
     write('prompt:');
     readln;
    end
  end.
```

The program above compiles and executes under Pascal–2 V2.1, but the program is not synchronized with input from the terminal. As a result, you do not see the prompt until you type in a line. In other words, you are prompted for the line you have just entered. For the prompt to be effective, program INTERACTIVE should be coded as follows.

```
program Interactive(input, output);
{sample for version 2.1}

begin
  write('prompt:');
  while not eof do
  begin
    readln;
    write('prompt:');
  end
end.
```

NOTE: Under the Digital operating systems, typing Control-Z (^Z) in response to the prompt signals an "end of file" on the interactive input file, halting the program.

Converting unsigned integers on the PDP-11

PDP-11 computers store integer variables in 16-bit words. These 16-bit words may be interpreted as signed or unsigned integers. As a signed number, in two's complement notation, 16 bits can represent numbers in the range of −32767..32767. When considered as unsigned numbers, a word can represent an integer in the range of 0..65535. For both signed and unsigned integers, the bit patterns in the word are the same; only the interpretation of the bit patterns differs.

When their values are compared or used in mathematical expressions, unsigned integers differ greatly from signed integers. As an example, consider a word in which all 16 bits are set to one. This word has a value of -1 when interpreted as a signed integer, or a value of 65535 interpreted as an unsigned integer. When this word is compared with some other value, the PDP-11 uses different combinations of instructions for signed and unsigned comparisons. If this number is multiplied by two, the result is a value of -2 for signed or 131070 for unsigned. The latter is an overflow condition because the result will not fit within 16 bits.

The Pascal-2 compiler and support library also differ in their treatment of signed and unsigned integers. The compiler can deal with unsigned integers, but the library contains a single routine that interprets all integers as signed values. For example, if you define a variable to be of type **integer** in your Pascal program, the compiler treats that value as a signed integer, unless you specify an unsigned integer using a subrange notation such as:

```
type
  unsigned = 0..65535;

var
  X: unsigned;
```

According to your data declarations, the compiler generates the correct code to compare, multiply, or divide unsigned numbers. However, the support library only prints out a signed integer unless you use the following procedure in your program instead of the **write** statement:

```
procedure Uwrite(X: unsigned;
                 Width: integer);

  var
    k: integer;

  begin
    if (X > 32767) and (Width >= 0) then
      begin
      if Width > 0 then
        Width := Width - 1;
        write(X div 10: Width);
        X := X mod 10;
        Width := 1;
      end;
    write(X: width);
  end;
```

This procedure takes an unsigned integer and a field width as its parameters. The number is printed as a value in the range of 0..65535, right justified in the specified field.

The PDP-11 floating-point hardware uses a signed conversion when it converts an integer value to a real value. If you wish to convert an unsigned integer to a real number, you should use the following function.

```
function Ufloat(X: unsigned): real;

  var
    R: real;

  begin
    R := X;
    if R < 0.0 then
      R := R + 65536.0;
    Ufloat := R;
  end;
```

This function takes an unsigned integer as its parameter and returns a real value in the range of 0.0 to 65535.0. The **trunc** and **round** procedures convert real numbers to integers. The floating-point hardware assumes a signed conversion, so the following function should be used when an unsigned integer result is desired:

```
function Utrunc(R: real): unsigned;

  begin
    if (R > 65535.0) or (R < 0.0) then
      writeln('Unsigned number out of range');
    if R > 32767.0 then
      R := R - 65536.0;
    Utrunc := trunc(R);
  end;
```

This function takes a real number in the range 0.0 to 65535.0 and converts it to an unsigned integer. The unsigned **round** function is very similar to the above unsigned **trunc** function.

The Log: Pascal-1 and Pascal-2

Pascal–2 V2.1 fixes a number of bugs, including many that generated obscure or random behaviors. Several fixes listed for V2.1 involve the support library; such fixes also apply to V1.3.

Pascal-2

V2.1 fixed the bugs consistently generating these symptoms:

/EIS, /FIS Arithmetic

Real operations under `/eis` and `/fis` generated incorrect results at times. Problem occurred in the handling of intermediate, stacked results of the simulated real arithmetic or (fis instructions) operations.

Packed Records

Packed records would generate incorrect results if the record was moved to the odd byte of the word.

Odd Address Traps

Several problems, resulting in odd-address traps at run time, were fixed.

Odd Address Traps During Set Operations

Compiler assumed that sets were always word-aligned (true unless the set is only one byte long). And it didn't ensure that the two operands were unpacked.

Illegal use of MOD

Illegal use of `mod` caused a compiler trap. An error message is now generated.

'%Include' Syntax Checking

Previously, the compiler didn't catch errors in syntax. It now generates the proper error message.

'Repeat .. Until' Errors

Errors were occasionally generated in `repeat .. until` operations.

Expression Evaluation Errors

During optimization, the compiler would sometimes evaluate an expression when it was not safe to do so.

Boolean Assignments in Debugger

If the first Debugger command in a program being debugged involved assigning a value to a boolean variable, the results would be "out of range."

Debugger 'L' Command

The Debugger L command was not ignoring enough lines at the top of each listing page; part of the page header was being included in the listing of the code.

Debugger, File Pointers

The Debugger had problems with pointers to files, either printing the wrong value or trapping.

Debugger, Packed Records

The Debugger was not making the transition between unpacked and packed parts of a record properly. Incorrect values were printed when individual fields were accessed.

The Debugger treated a packed array of unsigned bytes as if the array elements were signed, causing elements larger than 127 to print as negative.

Debugger, Real Constants

The Debugger gave incorrect results in writing the value of real constants.

/FTN Switch

When files were opened with the `/ftn` switch, the second character on the line was being used to position the cursor on CRTs. The fix moves the carriage-control character to a register to be sure that the upper byte is zero. A solution for users with previous versions is to write a null character `(chr(0))` as the second character on the output line, immediately following the FORTRAN control character.

Text Files and 'Put'

Creation of a text file with `put` operations, rather than `write` or `writeln`, altered one extra byte past the end of the allocated buffer. This would zap whatever data happened to be on the heap after the file buffer. The overflow was detected, but the heap was damaged.

Pascal-1

V1.3 bug fixes are currently in progress. A summary of the fixes will be included with the update release. Two fixed thus far are:

Abs() problem

The `abs()` function gave incorrect results for `real` operations. It now gives correct results for both reals and integers.

Trapping on underbars

A program containing underbars in identifier names caused the compiler to trap. It now gives an error message.

Pascal NEWSLETTER
OREGON SOFTWARE
2340 SW Canyon Road
Portland, Oregon 97201

Collins Hemingway, editor
David Spencer and Michael Kuhn, staff writers

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. We pay $100 per newsletter page for any article we print.

Printed in USA

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the Information Exchange. Your description should follow the format of the items below. Interested parties can contact one another directly.

Data Base written in standard Pascal. Intended as an educational tool and currently used by computer science classes, the working system consists of four interactive programs, with a screen updating program and a calculation routine scheduled for inclusion this summer. For information, contact: Leon Schilmoeller, Computer Science Dept, Augustana College, Souix Falls, SD 57102, (605) 336-5495.

Pascal Users' Group. The new contact person for the PUG is: Charles Gaffney, 2903 Huntington Rd., Cleveland, Ohio 44120.

Pascal Programmer sought by dynamic, research-oriented company. Ideal candidate should have a thorough knowledge of Pascal programming and be familiar with PDP-11 minicomputers and/or the Motorola MC68000 microprocessor. Send letter of inquiry and resumé to: Personnel Director, P.O. Box 1201, Portland OR 97207.

Compiler Writer wanted. Dynamic, research-oriented company seeks senior software engineer with a thorough knowledge of Pascal programming and an extensive background in writing code generators. Send letter of inquiry and resumé to: Personnel Director, P.O. Box 1201, Portland OR 97207.

OPUS Communiqué

Oregon Pascal Users Society

TUSTIN, California — As of February, Oregon Pascal User's Society (OPUS) has more than 100 members. At the first two OPUS meetings held in Irvine, California, the 10 attending members of the group decided that the organization should be structured as two major subdivisions: United States and International.

Within the United States, OPUS will be divided into subgroups by time zone, e.g., West Coast, Mountain, Central, and East Coast. The International OPUS will be subdivided by geographical proximity. The international group is not yet subdivided because of the small number of international members (so far) and their extreme geographic diversity. Canada, with 10 to 15 members, is an exception.

Many local OPUS (LOPUS) chapters can be formed within each time zone. Now, we need people in the local areas who will accept the responsibility of organizing a local OPUS. The Southern California OPUS (SCOPUS) is an example of the local OPUS structure that most members felt would be useful in attacking the problems of certain tasks they have encountered. The Southern California chapter held its first two meetings in January and February 1983.

Anyone interested in starting a LOPUS should contact Bruce Williams, the OPUS coordinator, at the address given below. He will help you get started.

OPUS Projects

OPUS is conducting a survey to determine the machines and compilers currently in use by its members and the tasks (business applications, systems, etc.) for which those systems are used. The questionnaire is being mailed to each OPUS member. Responses will be collected in March and April. After compiling the results, OPUS will send to each member a membership list and a copy of the results. With the results of the survey, an OPUS member can find out who in their area is using Oregon Software's Pascal and in what applications. If you aren't a member or if you have not received a member survey, please write to Bruce Bruce at OPUS.

We're also putting together a library of useful routines submitted by OPUS members.

Notes from OPUS Members

From EOCOM in Southern California — Recently, Lyle Norton discovered that the `time` function for RT-11 takes care of the 50/60 Hz conversion before it returns the time to the calling routine. You don't have to add your own conversion code. Isn't it nice to have things done right *before* you start the coding? You won't, however, find this information in the manual.

From Allergan in Southern California — Jerry Shaver has done some nice routines using character I/O on RSX-11M. Jerry said he would prepare them for the OPUS library being developed. If you're curious about these I/O routines, contact him through OPUS.

Next Communiqué

We'll say more about the OPUS library of routines and give you some guidelines for writing code that you want to submit to the library. If you want to appeal for a solution to some nagging problem or the ways others may have attempted to do something, please write to OPUS at the address below. We'll get your appeal into the Communiqué.

Bruce Williams,
OPUS Coordinator
c/o EOCOM
15771 Red Hill Ave.
Tustin, CA 92680
(714) 730-5051, ext. 302

Editor's Note:

The User's Manual for RT-11 says that the built-in function `Time` "returns a real value corresponding to the current time of day" on page 57. We saw no need to explain in detail how the time was produced.

Anne Smith retires; Pat Rau named manager

Anne Smith, Manager for General Distributors, retired on March 31. As a founder of Oregon Software, she has spent the last seven years helping the company grow and succeed, building up our distributor network and creating the sales group.

Retirement is probably the wrong word for Anne's future; redirection of her energy would be better. As Anne says, "There is so much to do that I always feel I need to rush out and do it all TODAY! That's one of the reasons for early retirement — to have the time to do it."

Travel is first on her new priority list. Anne plans a two-week cruise through the Caribbean islands, including two stops in South America. "I'm really going to tourist it," she says. Of course, she'll be doing a **lot** of fly-fishing on our beautiful western trout streams, and she'll be cross-country skiing in the winter, mid-week when the trails are not so crowded. She also wants to take a two-week raft trip on the Colorado River, through the Grand Canyon this year or next.

Anne wants to resume volunteer work for environmental groups such as the Nature Conservancy and Audubon Society. She used to be active in these groups, but hasn't had time during her years with Oregon Software. Her close friend Pete Pedersen, a lawyer, works as a volunteer for the ACLU and she may find something there, too.

From time to time, Anne will serve as a consultant to Oregon Software and she may fill in for vacationing sales personnel occasionally.

Pat Rau succeeds Anne as Manager for General Distributors. Pat has been with Oregon Software for two and a half years, as a general salesperson and specialist in licensing agreements. Before joining Oregon Software, Pat worked for Honeywell's Portland division.

Anne Smith

Pat Rau

David Cloutier heads marketing

David Cloutier

Oregon Software has appointed David Cloutier, formerly a software marketing manager at Intel, as vice president of marketing and sales.

David was the software marketing manager of OEM microcomputer systems at Intel's Hillsboro, Ore., division, and was product manager for the introduction of Intel's iRMX 36 operating system. Additionally, he served as chairman of the OEM division's software business planning committee. All together, David has 10 years of software marketing experience, having also worked at Zentec and Perkin-Elmer.

David expects Oregon Software to become a major software house for the MC68000, based on recent releases of a family of software products for developing high-performance applications for Motorola's MC68000. David also expects the company to continue its enhancement of products serving the DEC market, both through major upgrading of compiler products, such as the 2.1 release, and through new products such as SourceTools.

"The professional programmer is the primary focus of our work," David said. "That's because we are a company of professional programmers. Every tool we sell is a tool we first develop for our own use, and every tool we sell has to first meet our own high expectations. We must bring that quality advantage to bear in the market to give our customers the same high performance in their applications."

User manuals revised for V2.1/1.3

New editions of all PDP-11 manuals for Pascal–2 are expected to be published in late spring. New editions of Pascal–1 manuals will be printed in the fall.

The new editions will document all new software features included with the releases of Versions 2.1 and 1.3, respectively. The Pascal–2 manuals will include new sections, expanded sections, additional examples in many areas, corrections of errors and omissions, and inclusion of all current errata. The manuals also will have indexes.

Pascal–1 manuals will be upgraded to include many of the features now found in the Pascal–2 manuals, including tutorial sections and index.

Pending the printing of the new editions, the new 2.1/1.3 features will be documented in a supplement to the current manuals. The supplement will be shipped with all updates.

User manuals for all Oregon Software products will take a new form with all new printings: looseleaf with windowed covers and binders. (You remember the binders, don't you?)

This format will allow us to send send future errata as inserts, in the form of "change pages." Users can "mix and match" their Oregon Software library according to their choice of compiler and options. The manuals will now lay flat during use. The new cardboard cover allows users who preferred the bound editions to keep a semblance of that feel.

Book Prices

As of April 1, 1983, new prices will be in effect for all books and manuals available from Oregon Software. The new price list follows:

User Manual	Type of Sale	Price
Pascal-1 and Pascal-2	End User	$15.00
	Volume Purchases/Instructional	$10.00
SORT-1-Plus	End User	$10.00
	Volume Purchases/Instructional	$ 7.00
SourceTools	End User	$10.00
	Volume Purchases/Instructional	$ 7.00
Concurrent Programming Package	End User	$15.00
	Volume Purchases/Instructional	$10.00

Book	Author(s)	Price
Algorithms + Data Structures = Programs	Wirth	$27.95
Elements of Programming Style	Kernigan, Plauger	$14.95
Introduction to Pascal	Zaks	$15.95
Pascal User Manual and Report	Jensen, Wirth	$ 9.50
Programming in Pascal	Grogono	$18.95
Structured Programing	Dahl, Dijkstra, Hoare	$20.00
Systematic Programming: An Introduction	Wirth	$26.00
Tex and Metafont	Knuth	$12.00
Concurrent Euclid, The UNIX System, and Tunis	Holt, Graham, Lazowska, Scott	$15.95

Books and manuals are shipped FOB destination (within USA) or FOB Portland (outside USA).

2340 SW Canyon Road, Portland, Oregon 97201 Printed in USA

Pascal
NEWSLETTER

NUMBER 7 OREGON SOFTWARE SUMMER/FALL 1983

UNIX compiler for 68000 released

At the USENIX Conference in Toronto, Oregon Software introduced the first high-performance Pascal compiler for the growing UNIX market on the Motorola MC68000. The compiler is available by itself or with the source-level, interactive Debugger and other software development utilities.

UNIX, developed by Bell Labs and licensed through Western Electric, is becoming a major development system on the 68000. Several dozen vendors are now offering UNIX or UNIX-like systems for applications development on the MC68000. Pascal–2 offers the benefits of Pascal without sacrificing the code quality required in the UNIX production environment.

Like other Pascal–2 products, the UNIX compiler performs eight optimizations on user programs, producing small, fast code. Oregon Software's benchmarks show that code produced by the Pascal–2 compiler is more compact and efficient than that of other high-level languages on 68000 UNIX systems, including C, and is an order of magnitude faster than interpretive or threaded languages.

Debugger improved

A key feature of UNIX is the concept of supporting "tools" — standard, easy-to-use programs that aid a programmer in coding and development. Like Oregon Software's existing Pascal packages, Pascal–2 for UNIX contains a set of development tools — a debugger, execution profiler, program and text formatters, and cross referencers for program analysis.

As with the other Pascal–2 debuggers, the UNIX debugger allows the programmer to interactively solve logic errors in applications at the level of the source program, thus simplifying and speeding development. The UNIX debugger, however, runs as a separate process from the application code being debugged, keeps track of multiple compilation units, and performs breakpoint debugging without slowing down the application code.

The profiler and other utilities perform substantially the same as they do on other systems.

Product unbundled

Pascal–2 for UNIX may be purchased as a complete package. Users also have the choice of purchasing the compiler only, and may opt for short-term or annual support.

Documentation includes the *Pascal–2 User Manual* with User's and Programmer's Guides, Debugger and Utilities Guides, and a detailed Language Specification; inserts for the UNIX Programmer's Reference Manual; and two textbooks on programming in Pascal.

Version 2.1B release nears

Pascal–2 Version 2.1B is currently in field test. Release to general users is scheduled for Nov. 1. Numerous V2.1A bugs have been fixed and the ability to debug external procedures has been added. All supported users may request updated software and documentation from Oregon Software or their distributor.

In this issue...

UNIX/68000 compiler released . . . . . Page 1
RSTS SourceTools available . . . . . Page 2
Pascal–2 on DEC Pro 350 . . . . . Page 2
New releases for VAX, VERSAdos . . . . . Page 3
V2.1B ready . . . . . Page 4
Using unformatted files . . . . . Page 4
Letters . . . . . Page 6
OPUS Communiqué . . . . . Page 7
Accessing global symbols . . . . . Page 7
Pascal–2's concurrency package . . . . . Page 8
Pascal–1 real-time facilities . . . . . Page 11
The Log . . . . . Page 18
Errors, additions to manuals . . . . . Page 20
Information exchange . . . . . Page 21
Pascal standard . . . . . Page 21
Sales staff . . . . . Page 22
Contest winners . . . . . Page 22
Transition marked . . . . . Page 23

Source Tools also available for RSTS/E

Oregon Software's SourceTools, a software management system, is now available for the RSTS/E operating system. The product was earlier released for the VAX/VMS and RSX-11M operating systems.

SourceTools allows programmers to manage access to sources and to document changes in sources over the development life cycle of a product.

In the past, software developers have relied on informal, manual methods to manage software development. With software projects growing in complexity and development teams increasing in size, these traditional methods of project management have eroded the real productivity of programmers.

SourceTools addresses this problem by providing an effective, automatic management scheme for software projects, particularly those involving numerous source files, joint development by several programmers, or the creation of cross-system software. SourceTools may be used with software developed in any computer language and with any standard text file.

The core of SourceTools is a package called SourceCon that controls the creation and modification of source files. Another program, MAKE, automatically keeps programs up to date as components change. Two other programs, TXTCOM and SEDIT, function together to ease the task of maintaining parallel sources on different systems.

Control of changes provides key

Controlling access to a file, and recording any change to it, is the essence of a source-control system. With SourceCon, a file placed under source control becomes the base version. As the software is developed, each change to the base file is recorded separately, with a clear description of the person making the change, its purpose, the date and time, and the actual textual change.

Any version may then be reconstructed. Developers are able to generate releases from source-controlled files while simultaneously developing later versions of the same files. Developers are able to control the branching of software or documentation into several similar products without a proliferation of redundant files.

In addition, the clear audit trail helps developers isolate and correct human errors. In the case of large-scale mistakes, developers may replace the latest version of a file with an earlier one. Further, only one developer at a time has "write" access to source-controlled files, preventing conflicting changes from being made — a common cause of confusion and product delay in multi-programmer development.

'MAKE' rebuilds programs

A second component of SourceTools is MAKE, a file rebuilder that prevents the accidental inclusion of outdated modules into a large program compilation. After reading a user-written file describing the dependencies of each module on another, MAKE checks the date and time of each module and automatically rebuilds any that are out of date with respect to others. This facility improves program reliability and frees programmers for other tasks.

MAKE minimizes the number of commands required to rebuild programs and makes explicit the file dependencies.

Maintenance eased

The third major portion of SourceTools consists of two programs that, used together, automate the maintenance of parallel source files. TXTCOM compares the contents of two files and generates a script of the differences between them; SEDIT, a stream editor, reads the TXTCOM edit script and applies the necessary changes to one of the parallel source files. Because only the script of differences has to be transferred, these programs minimize the time required for updating sources at remote sites or on different processors.

SourceTools is available from Oregon Software or its authorized distributors. Documentation includes a 75-page user manual.

Pascal-2 moves to DEC's Pro 350

Beginning April 1, Oregon Software and authorized distributors are offering the Pascal–2 optimizing compiler for the DEC Professional 350, Digital's most advanced personal computer.

Developers may write Pascal–2 programs directly on the 350 under the RT-11 system. They also may use the 350 as a target machine, writing programs on the VAX or the PDP-11 and transfering the code to run on the 350.

The Pro 350 is a familiar environment for professional programmers. Many of them were trained on the 350's underlying PDP-11 hardware; Pascal–2 is the most widely used Pascal on the PDP-11. This means that a large body of applications programs may be moved to the 350 rapidly.

Standard language allows integration

The Pascal–2 language implementation is standard across all Digital operating systems, plus UNIX. Pascal–2 for the Pro 350 supports all capabilities of standard Pascal, including conformant array parameters, and code developed under Pascal–2 may be moved to other operating systems that have standard Pascal compilers.

Pascal–2 offers low-level integration with the RT-11 system. Pascal–2 programs may call subroutines written in Pascal or assembler, allowing the user to take advantage of existing system software. Pascal–2's language extensions provide sophisticated I/O handling and access to system-specific operations.

Tools included as options

Pascal–2 for the 350 includes, as an option, the full set of supporting tools available on other Pascal–2 systems: an interactive source-level debugger, an execution profiler, program and text formatters, and cross referencers for program analysis.

License fees for the complete Pascal–2/350 system are: $450 for the compiler and support library, including 90 days' warranty, documentation, sample program, and subscription to our newsletter. A package containing the debugger and utilities is $350. Software updates are $300 per update. All prices are U.S.

Documentation includes the standard *Pascal-2 User Manual.*

VAX cross-compiler targets any 68000 hardware, includes real-time capability

Oregon Software's 32-bit, VAX-based Pascal–2 system for developing applications software to run on the Motorola MC68000 is now available.

The cross-development software allows users to target applications, including real-time programs, for virtually any 68000 hardware configuration. Most of the development effort takes place in Pascal, with development on the 68000 limited largely to final testing.

The cross-development system consists of the optimizing Pascal–2 cross-compiler, which supports the international standard; a library of routines allowing user programs to run without an operating system on the 68000 and providing the capability for concurrent programming; a cross-linker/assembler producing loadable code for the 68000; and program development utilities such as PASMAT and XREF.

The VAX/VMS cross-compiler supports 32-bit integers. A 16-bit version was released earlier for RSX on the PDP-11.

The Pascal–2 cross-compiler performs extensive error checking and uses the same optimization techniques as the other Pascal–2 compilers to produce small, fast code. Pascal–2 supports conformant array parameters, adhering to Level 1 of the international Pascal standard.

Concurrency supported

Control of real-time processes is offered via the Concurrent Programming Package, which is an option to the system. The package enables users to develop all parts of an embedded or stand-alone system, even device drivers, in standard Pascal. The package offers true priority scheduling with a minimum of overhead.

The package consists of a library of procedures and functions that allow concurrent programming in the "process-monitor" style. This allows you to apply structured programming techniques to the problem of synchronizing the exchange of shared data among separate, concurrently executing processes.

The library also provides the interface with the hardware so that programs may run without an operating system. The library can be adapted to any hardware configuration; source code for all modules is provided to allow maximum flexibility when implementing the system.

Available now

The development system is now available from Oregon Software or its authorized distributors; the exact price depends upon the options selected.

32-bit VERSAdos native announced

In conjunction with release of the cross-compiler systems, Oregon Software also is announcing a native VERSAdos 32-bit compiler for the MC68000. This package includes an interactive source-level debugger, an execution profiler, and the other standrd Pascal–2 utilities such as formatters and cross-referencers. The compiler generates code compatible with the Motorola linker/assembler and also has the Concurrent Programming Package as an option.

How to read/write unformatted FORTRAN files from Pascal-2 under RSX-11M

by Joerg Donandt and Gerald Jahn

FORTRAN unformatted files are an efficient means of storing large data structures with respect to both storage and execution time. Unformatted data files require less storage space because redundancy is minimal. Conversions between internal binary form and a form readable by man (e.g., ASCII format) are unnecessary. For these reasons, we opted to use FORTRAN unformatted files to store digitized pictures, but we found that we needed to find a way to read and write them from Pascal programs — specifically those compiled with the Pascal–2 compiler. This article contains our solution and two samples of its use in Pascal programs.

Unformatted file structure

The way data are stored and accessed in unformatted FORTRAN files causes the difficulty. Data are stored in variable-length records (132 bytes maximum length) that may wrap around block boundaries. Normally, carriage-control characters determine the way that the FORTRAN files are printed out, but the file management system cannot distinguish between a normal FORTRAN print file, generated using a `WRITE(6,100)`... statement where 100 is the format statement for the write, and an unformatted file generated by `WRITE(3)`... statement.

Access to these records depends upon a two-byte length counter, which isn't normally available to the Pascal programmer. When accessing these records, the 132 bytes are read into the run-time system's buffer as a "segment," and every segment begins with a two-byte segment descriptor that indicates the segment's place in the file by one of four values:

Value	Description
1	first segment of the file,
2	last segment of the file,
3	only segment of the file,
0	any other segment between the first and last segments of a file.

Every filled record holds 130 data bytes, 2 segment-descriptor bytes, and 2 length-counter bytes, for a total of 134 bytes per record.

In the 130 bytes following the segment descriptor, data are interpreted according to the variable-type and the sequence in which those variables were written into the file. For example, a FORTRAN expression such as `INTR*2-VALUE` is stored as two data bytes, with the byte representing the least significant digits first.

Multidimensional arrays, such as those in the sample programs, are stored in a "first-index-quickest" fashion: the elements of an array are stored in adjacent locations, the first index is incremented for each element stored until its range is exhausted, then the second index is incremented. Thus, a matrix `MAT(I,J)` is stored with the elements indices, as shown:

```
(1,1)(2,1)...(I,1)
(1,2)(2,2)...(I,2)
  ⋮     ⋮       ⋮
(1,J)(2,J)...(I,J)
```

The storage required to hold a data item such as our picture-matrix (named `Pic` in the two sample procedures) would have to be declared `DIMENSION BYTE MAT(128,128)` consisting of 128 * 128 bytes = 16384 bytes, stored in 130-byte segments. The file consists of 126 filled segments and one 4-byte "appendix," for a total of 127 segments.

For such calculations, you need to be aware of the following numbers: every filled record holds 130 data-bytes, 2 segment-descriptor bytes, and 2 length-counter bytes, for a total of 134 bytes per record. The last record consists of 4 data-bytes and a 4-byte overhead, for a total of 8 bytes. The file's length is calculated as:

$$134\ \textit{bytes/record} * 126\ \textit{records} + 8\ \text{bytes} = 16982$$

These are stored in 33 decimal blocks (44 octal), with each block containing 512 bytes, leaving 4 bytes empty in the last block.

Reading unformatted files

In order to read unformatted FORTRAN files, you must know **how** the unformatted file was written. With this in mind, look at sample procedure 1 (**Pic_read**). The file was generated by the FORTRAN statements:

```
DIMENSION BYTE MAT(128,128)
    :
WRITE(3) MAT
    :
```

From the above we know that:

- The file holds only one variable: **MAT**.
- The variable is structured as a two-dimensional array, with indices ranging from 1 to 128.
- The elements of this array are byte-values: they require only one byte storage per element.
- The elements are stored in standard form as previously explained.

To express these storage and structure characteristics in Pascal, we define the segment as:

```
type
  Byte = 0..255;
  Pic = packed array[1..128,1..128] of byte;
```

Note that we use a packed array. Since **Byte** is a subrange of **integer**, each element of **Pic** would normally be allocated two bytes of storage because **integers** are stored in two bytes. The packed array, however, forces the compiler to allocate only the storage needed, in this case, one byte.

The procedure does three basic operations: opens the appropriate unformatted file; reads a record from it; and transfers the data bytes into the **Pic** matrix. The procedure repeats the loop of reading and transferring until the matrix is filled.

The **reset** statement must contain the following file attributes:

/VAR:132 Variable-length record (maximum 132 bytes).

/NOBLK Records may wrap around block boundaries.

/FTN Use FORTRAN's carriage-control convention.

In **Pic_read** we have used the **read** statement for non-text files, violating the standard. The end-of-file (**eof**) does not work with non-text files, so we have introduced our own marker (**Endf**).

The statement **case** will handle the different possible segment-descriptors (**Code** in this procedure). The statement **code: for 3** would be somewhat inconsistent with our knowledge of **MAT**. Since **MAT** is large (about 16K Bytes), it must require more storage than a single record can provide. **Pic_read** therefore aborts for **Codes** not in the range 0 to 2. The **FOR K:=1 TO 130** loop empties the **Data** section of the segment, byte by byte, and stores the data at the appropriate position (indexed by **I,J**) in the **Matrix**.

The procedure is straightforward, and, once you've understood the details, you can modify it easily to fit your needs.

A tricky approach, however, must be used in writing unformatted FORTRAN files from Pascal.

Writing unformatted FORTRAN files

When we first tackled this problem, we had extreme difficulties writing segments into a variable-length record, non-text file. Indeed it was impossible. So we circumvented the problem with the following trick.

Instead of writing to a non-text file of variable-length records, we specify a normal text file and, using the **chr** function, we suppress the formatting normally done by the **write** statement. In addition, we used the **/FTN** attribute so that the FORTRAN run-time system (FOROTS) can find the attribute set when it tries to read unformatted data from the file.

Sample procedure 2 (**Pic_write**) does not write the file in portions of "records" but byte by byte. It starts by writing the segment-descriptor as two bytes (low byte first). Then it fetches a byte from the matrix (indexed by **I,J**) and writes it to the file. When it has written 130 bytes, a **writeln** lets the system store the record away and start a new one. During the **writeln**, the file buffer is emptied and the length-counter (mentioned in the preceeding discussion of file structure) is written to the file. Earlier, the length-counter could not be written, probably because the programmer is not allowed to use **writeln** statements on non-text files.

If you need to read/write unformatted FORTRAN files from the Pascal environment you ought to have a sound knowledge of the way both compilers treat your data. Although a general solution cannot be given, we hope that these remarks can help you solve your problem.

Mr. Donandt and Mr. Jahn are engaged in optical/digital image processing for pattern recognition purposes at The Institute of Applied Physics 1, University of Heidleberg, in Germany.

Sample Procedure 1

```
procedure Pic_read(fname: string;
                   var matrix: pic);

  type
    vlr = record   { a FORTRAN variable-length record }
      code: integer;
      data: packed array [1..130] of byte;
      end;

  var
    i, j, k: integer;
    datei: file of vlr;
    nxtrec: vlr;
    endf: boolean;

  begin
    endf := false;
    reset(datei, fname.ch, '/var:132/noblk/ftn');
    while not endf do
      begin
      read(datei, nxtrec);  { non-standard read }
      with nxtrec do
        begin
        case code of
          0: ;
          1:
            begin
            i := 1;
            j := 1;
            endf := false;
            end;
          2:
            endf := true;
          end;  { abort, if another CODE occured }
        for k := 1 to 130 do
          if j < 129 then
            begin
            matrix[i, j] := data[k];
            i := i + 1;
            if i > 128 then
              begin
              i := 1;
              j := j + 1;
              end;
            end;
        end { of with } ;
      end { of whilf } ;
    close(datei);
  end;
```

Sample Procedure 2

```
procedure Pic_write(Fname: string;
                    var Matrix: pic);

  var
    i, j, k: integer;
    datei: text;
    endf: boolean;
    c: byte;

  begin
    endf := false;
    i := 1;
    j := 1;
    rewrite(datei, fname.ch, '/var:132/noblk/si:-1/ftn');
    while not endf do
      begin
      c := 0;
      if (j > 127) and (i > 124) then
        begin
        endf := true;
        c := 2;
        end;
      if (j = 1) and (i = 1) then
        c := 1;
      write(datei, chr(c), chr(0)); {segment descriptor}
      for k := 1 to 130 do{ fill a record into MATRIX }
        begin
        if j < 129 then
          begin
          write(datei, chr(matrix[i, j]));
          i := i + 1;
          if i > 128 then
            begin
            i := 1;
            j := j + 1;
            end;
          end;
        end;
      writeln(datei);
      end;
    close(datei);
  end;
```

Letters

To the Editor:

The following is a plea to whomever (or whatever) has control over the release of existing errors in the Pascal-2 compiler. I realize that there may be a bit of pride which must be swallowed to admit that there is a bug in one's software but it must be done. Anyone familiar with the complexities of a compiler, much less one that performs such significant optimization, realizes that such bugs must exist but should be able to accept that.

As a user of Pascal-2, I would have to say that finding a compiler error is very frustrating. It is not until I have spent several days tracing down the bug, only to find that when I call one of your very helpful software support personnel that it is a bug which has been reported and known about, that I become very disappointed in the product. If I felt that I was the first person to encounter the error or had some chance to avoid it had I known about it, I would feel much more forgiving. (Obviously not admitting that it has been reported and resolved would be unforgivable.)

Of course we would all like every software product to be perfect but it isn't, not yet at least. So why kid ourselves? Let's communicate. The savings in real dollars as well as goodwill, I feel, would greatly outweigh the pride and postage it might cost you.

For instance, a compiler error could take as much as a day to track down to determine that it is in fact the

compiler and not a program error. If a programmer is paid nine dollars an hour and puts in an eight-hour day tracking the error down, it would be 72 dollars wasted. Multiply that by 100 installations which might come across the same error and already you have potentially wasted $7,200. Don't forget to take into consideration the friction created between the EDP managers, their programmers and Oregon Software.

All this could have been avoided with a simple monthly letter to all supported sites which would describe the problem and the way, if any, to code around it. I realize that compiling such a letter may be non-trivial but its benefits out here in user land would outweigh that significantly.

One other request I have is for a fast single-pass version of the compiler with, perhaps, optional code gereation. This again would create a great savings for us out here who must wait for 8 minutes, and sometimes more, for compiles on our meager 11/34s only to discover a missing variable declaration.

Both of these requests I make with all sincerity and hope they will be considered with the same degree of seriousness.

Sincerely, Harry A. Levinson

Editor's Reply:

Your plea for notice of known bugs has been heard, and we are in the process of improving our response. We are now better able to describe and track reported errors in a useful manner and hope to find a quick and painless way for users to keep up to date. We'll print a statement of the new policy in a future issue of the newsletter.

We have no plans for a single-pass version of the Pascal–2 compiler.

OPUS Communiqué

Oregon Pascal Users Society

Members should have begun receiving their membership list and member survey sheets on October 14, 1983 — just in time to contact others in the OPUS for Halloween.

The OPUS library will be opening soon. The logon for the OPUS library will be:

```
HELLO OPUS/LIBRT
    or:
HELLO OPUS/LIBRSX
    or:
HELLO OPUS/LIBRSTS
```

When you log on, a message appears:

```
For more information,
type SYSMSG.LST.
```

This file contains enough information to get you started on using the OPUS library. The OPUS library will be a read-only UFD. Dial-up lines are being installed now and the numbers should be in the October membership packet.

To put a program in the OPUS library, you must submit the source file only, with documentation of what it does, how to use it, how to compile and task-build/link it, and what its inputs and outputs are.

Everything submitted will be put into the library as long as it is consistent with OPUS goals and concepts. Send your routines to:

Bruce Williams,
OPUS Coordinator
c/o EOCOM
15771 Red Hill Ave.
Tustin, CA 92680
(714) 730-5051, ext. 302

Some have expressed interest in having a DECUS-like meeting. Such a meeting is really time-consuming and requires a lot of effort. All that can be said at this time is maybe and only if there is more interest.

In the meantime, we will "nybble" on the membership list problem. More information will be in the next Communiqué.

Accessing global symbols from Pascal programs

By Steve Poulsen

Many customers have asked how to access global symbols from a Pascal program. These are the global symbols used by the Linker and Task Builder, not the global-level variables defined at the start of a Pascal program. Users commonly want a data table defined in an assembler program from Pascal. It can be done, but it is tricky.

The concept of external data does not exist in Pascal, but the concept of external procedures does. The compiler generates a reference to the entry point of an external

procedure, using the first six characters in the procedure name as a global entry point. We use the compiler's ability to generate the address of a procedure to obtain the address of an external procedure.

When a procedure is passed as a parameter to another procedure, two parameters are passed. The first parameter is the static link used to access intermediate-level data in the proper context by the called procedure. Don't worry about it, we won't need to use it. The second parameter is the actual address of the procedure in memory. This is what we are after. Now all we need to do is to obtain the value of such a procedural parameter.

You must use an external procedure to obtain these two parts of a procedural parameter without violating Pascal's type checking rules because parameter typing cannot be performed across module boundaries. Define a module called GETADR.PAS (see example), a very simple function that takes two integer parameters and returns the second parameter as the function result. The trick is that we are really going to pass it a procedural parameter, and it will return the address of the entry point of the procedure. This function must be compiled as an external procedure to avoid the compiler's type checking rules.

The secret of accessing global symbols is to define the symbol as an external procedure and pass that procedure to a function that returns the address of the variable. For example, suppose that you want to access the variable **$DSW**, which is the RSX directive status word whose location is defined by the Task Builder. You include the function **Getadr** in a program as shown in the figure below. When the program is compiled and linked with the GETADR module (shown above the program), it prints the current value contained in the global variable **$DSW** (usually a 1).

```
{$nomain}
program Getadr;
{  Return the address of a procedure}

  function Getadr(static_link, entry_point: integer ): integer;
  external;

  function Getadr;
    begin
      Getadr := entry_point;
    end;


program Print_Dsw;

type
  Pointer = ^integer;

var
  Dsw_pointer: pointer;  { Address of DSW }

procedure $Dsw;
  external;

function Getadr(procedure Magic): integer;
  external;

begin
  Dsw_pointer := Loophole(pointer, getadr($dsw));
  writeln(''$Dsw = '', DSW_pointer^);
end.
```

Meeting design goals for Pascal-2 Concurrent Package

By Michael S. Ball

As computers increase in power and drop in price, many individual processors are being included as components of larger systems. These component processors are called embedded systems, and they are typically used for process control, data logging, communications processing, and signal processing. Such computers must deal with unique I/O devices and frequently must respond to external events in a short time. Traditionally, such applications have been coded in low-level languages, usually the assembly language for the machine. A part of the same tradition recognizes that such applications are difficult to code and even more difficult to debug. An efficient approach to concurrency is crucial to a successful embedded system.

Concurrency models

Two basic models are used in writing concurrent programs. A "message model" looks at a concurrent program as a collection of processes connected by message queues. The processes communicate by passing messages between themselves. A "procedure model" sees a concurrent program as a group of processes that communicate through procedures and shared memory. The two views are in fact isomorphic, and any concurrent program written using one viewpoint can be rewritten using the other.

The difference appears to the programmer in the "primitives" supported by the systems. The primitives of a message-passing system are usually variations on two types: "send a message" and "receive a message." Special-purpose primitives may also exist, such as "send a message and wait for a reply," but the only required primitives are "send" and "receive." The primitives in a procedure-based system

are "acquire access to shared memory" and "release access to shared memory" (historically called "P" and "V"). Again, specialized versions may be implemented, but the only neccessary ones are "acquire" and "release."

Each set of primitives may be implemented in terms of the other set and neither has any "special ability" not shared by the other. So how do we pick the approach to use? We look at the environment and applications intended for the system.

Let's consider two hypothetical examples. The first system consists of a number of microprocessors, each containing its own memory. Each microprocessor executes a single process, and the processes communicate over I/O channels. In this system, a message-passing scheme maps directly onto the hardware and is the obvious choice. The procedure-based system would have to simulate shared memory with messages, a profitless venture. In the second system, one or more processors share a common memory. Each processor may execute one or more processes, but large amounts of data are passed between processes. A procedure-based approach would be more appropriate in this system because communication takes place using shared memory anyway and a message-based system requires copying the data as it passes between processes.

An additional factor to consider is the type of I/O devices to be handled. The usual minicomputer or microcomputer device is easily treated as a process that shares memory with internal processes and communicates using signals. Devices with separate channel controllers may well fit more easily into a message model.

Our design goals matched better with the procedure model, so the Concurrent Programming Package supports that model of concurrent programming. The process-monitor approach of the Concurrent Package is based on a model that is slightly more refined than the basic one described above.

Process-monitor model

Process-monitor style concurrent programming was initially proposed by C. A. R. Hoare [1979] and has been used in the programming languages Concurrent Pascal [Hansen 1977] and Modula [Wirth 1977].

The objective is to isolate all communication between processes into *monitors*. Each monitor consists of some shared memory (monitor variables) for communication and a set of procedures to manipulate this shared memory. Each queued procedure executes an "acquire access" operation on entry and a "release access" operation just before exit. All manipulations of the shared memory must take place within these monitor procedures, guaranteeing consistency of the data.

If a procedure within a monitor has to wait for some external condition, it does a "release access" operation for the monitor variables and waits — using a variation of "acquire access" — for some other process to make the condition true.

Although monitors need only "acquire" and "release," specialized forms of these operations improve efficiency and programming ease.

The Pascal–2 Concurrent Programming Package uses two special operations, **lock** and **unlock**, to control access to monitor variables. Each monitor has a special variable, a *gate*, that is used as an argument to **lock** and **unlock**. The monitor code calls **lock** on entry to the procedure and **unlock** on exit. Without a separate gate for each monitor, entry into any monitor would lock out entry into all other monitors.

The special operations **wait** and **signal** provide synchronization between processes. The process that executes **wait** unlocks the monitor variables and suspends execution until a **signal** is executed by some other process. **Wait** and **signal** operations are controlled by special variables called *events*. One speaks of "waiting for an event" or "signaling the occurrence of an event."

Device interface

Embedded systems frequently have to drive a variety of I/O devices, and a useful concurrent programming system must provide support for device drivers. A device handler requires three things of the system:

• Access to the control registers for the device. On DEC's PDP-11 and Motorola's MC68000, device registers are mapped into the address space and manipulated like any other location in memory. The **origin** feature of the Pascal–2 compiler may be used to place a variable at the physical address of a device register; the register is then manipulated with normal assignment statements. Some additional procedures may be necessary for a computer that uses special I/O commands.

• Synchronization with the external device. Since external devices provide truly concurrent execution, even in a single-processor system, the hardware usually provides interrupts to signal that execution is completed. We can make the interrupt perform a "signal" operation as though it were an internal process.

• A way of providing mutual exclusion for access to the shared variables. The usual way of handling this in a computer is to turn interrupts off when manipulating data that is shared with the external device.

The Concurrent Programming Package meets the last two requirements with two new primitives. The first one is **makedevice**, which takes a gate as an argument and specifies

that the calling process is to execute with interrupts off. In addition, any process executing a **lock** on that gate has interrupts off until it executes a **wait** or an **unlock**. The process that calls **makedevice** is called a "device process."

Any device process can execute the **intwait** primitive, which has the same effect as a **wait** except that the signal is provided by an external device. During the time a process is waiting for an interrupt, other processes may execute within the monitor and have access to the shared memory.

The device process is sometimes called a "shadow process" for the external device. The combination can be considered a single process that executes on the external device during the **intwait** period and on the central processor at all other times.

Implementation decisions

The simplest way to achieve concurrency on a single processor is the use of coroutines. The processes in a coroutine system execute one at a time, changing control only at specific points in the code. Using of the primitives described above, the control changes from one process to another only at calls to **wait** and **signal**. **Lock** and **unlock** primitives become unnecessary, as control changes only when the monitor gate would be unlocked anyway.

Device processes can be incorporated in a coroutine system quite simply. The **intwait** primitive saves and restores registers so that the execution of the device process has no direct effect on other processes. When the device process executes a signal that actuates an ordinary process, control does not pass directly to that process. Instead, the process is made ready and control passes to it at the next **wait** operation performed by an ordinary process.

The coroutine system is attractive in its simplicity. The overhead is minimal; a process swap costs little more than an ordinary procedure call. If the primitives are built into the compiler, the routine approach doesn't even need to save registers across a swap. On the other hand, response to I/O is delayed until a process voluntarily relinquishes control. A task with long periods of computation between interactions can slow response drastically.

A full concurrent programming scheme should also provide for process priorities and preemption of running processes. In such a system, process swapping occurs whenever a process with a higher priority than the current process becomes ready to execute. The disadvantage of such a system lies in the increased frequency of process swapping and the added complexity of each swap. Careful implementation can minimize the frequency of process swapping, however, and the added complexity is a small price to pay if you need to combine long computations with fast response.

Our implementation

The Concurrent Programming Package provides Pascal–2 programmers with a set of primitives for programming embedded applications. The package supports multiple processes, device interfaces, and synchronization of processes. The entire application, from device drivers up, can be coded in standard Pascal.

The design goals for the package were to provide:

- Support for a variety of embedded systems. This includes a potentially large number of concurrent processes, some of which may have long sections of computation with interactions between other processes.
- Resultant software or programs able to run on a single mini or micro. Multiple processor systems are not difficult, but the coding details are dependent on the exact architecture of the system.
- Ability to write system-specific device handlers in Pascal.
- Minimal response time to external events and reduced cost of interprocess communication, so that it is cheap enough to be used whenever it fits the application.
- Resulting code capable of operating in a mixed ROM/RAM environment.
- A small, modular package. If an application does not require a particular capability, it should not have to pay the price for it.
- A self-instrumenting package that gives the programmer complete information on the state of the system when an error occurs.
- The ability to gather sufficient data on a program to spot and eliminate bottlenecks.

The Pascal–2 Concurrent Programming Package provides a scheme for multiple process priorities together with preemption. The extra cost is small compared to the capability gained. Process swapping is extremely efficient, with about 20 instructions executed in the worst case. This is little more than a simple procedure call.

The package includes a comprehensive set of diagnostic tools, including a process-by-process walkback in case of error. A log of the most recent primitive calls aids in the detection of subtle synchronization problems. The statistics gathered for each gate show the number of times the gate is used and the number of conflicts arising from two processes attempting to use the same monitor at the same time. Such statistics allow the detection of major performance bottlenecks.

The package meets all of the design goals and provides a useful addition to the software toolbox of system developers.

References

C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, Vol 17, No. 10, October 1974.

Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

N. Wirth, "Modula, A Language for Modular Multiprogramming," *Software Practice and Experience*, Vol 7, No. 1, 3-35 (1977).

Mr. Ball designed and implemented the Concurrent Programming Package at Oregon Software. Upon leaving Portland for a sunnier clime, he founded TauMetric Corporation, where he is a consultant for custom system software development. Address: 1094 Cudahy Place, Suite 302, San Diego, CA 92110, (619) 275-6381.

Implementing real-time facilities in Pascal

by Hilding Elmquivst and Sven Erik Mattsson

Introduction

The increasing use of microcomputers in technical systems has made the art of real-time programming more important. We have previously used Concurrent Pascal [Brinch Hansen, 1971] in courses on real-time programming. However, we needed more flexibility. For example, we wanted to be able to demonstrate message-passing schemes and rendezvous. Furthermore, in order to be able to give the students a profound understanding of how concurrency is achieved, the nucleus or kernel of such a system has to be shown.

A natural way of introducing concurrency is to start with a sequential language such as Pascal and add necessary routines without changing the compiler or the support library. Per Brinch Hansen [1978] and Kriz and Sandmayr [1980] give descriptions of such routines in Pascal-like notations. However, it is not always possible to write these routines in Pascal because hardware facilities such as registers must be manipulated when the processor is switching between processes. Brinch Hansen [1978] translated the kernel by hand to assembly language.

The real-time kernel we've developed supports concurrent programming in Pascal, particularly in Pascal–1. The kernel is written in Pascal, but it relies on a small number of assembly-language procedures for process creation, transfer between processes, and handling of interrupts. This set of routines is called the nucleus. The kernel implements semaphores for mutual exclusion and events for other synchronization. The kernel also provides the capability to program I/O handlers. With this real-time kernel, Pascal has the same expressive power as Concurrent Pascal. However, since the user's concurrent program is compiled with the standard Pascal compiler, there is much less security than in Concurrent Pascal.

In this article, we've described the implementation for Pascal–1 on the PDP-11 computer. The kernel has also been implemented on the Texas Instruments TMS 9900 and Motorola MC68000 processors.

Kernel primitives

Three primitives handle creation, scheduling, and execution of processes. Others perform duties such as communication between processes and synchronization of their execution, timer control, and interrupt handling.

Process handling

A process is declared as a parameterless procedure, which we refer to as "process-procedure" hereafter.

A process instance may be created by a call to `CreateProcess` from anyplace where the procedure could be called in the ordinary way:

```
procedure CreateProcess
  (procedure proced;
   memreq: unsignedinteger);
```

where *proced* is the process-procedure describing the process, and *memreq* is the memory requirement (in bytes) for the stack and the heap of the process.

The processes are scheduled according to their priorities. The priority of a process can be changed dynamically by calling:

```
procedure SetPriority
  (priority: integer);
```

where *priority* is the new priority of the process.

The main program must be converted to a process by a call to `InitKernel` before other processes can be created:

```
procedure InitKernel
  (memreq: unsignedinteger);
```

where *memreq* is the memory requirement (in bytes) for stack and heap (global variables excluded). The procedure `InitKernel` also creates a clock process and an idle process

Communication

Communication of data between processes is done with variables that are accessible to the process-procedures according to scope rules. The programmer must ensure mutual exclusion by the use of semaphores, as follows:

```
procedure InitSemaphore
  (var sem: semaphore;
   initval: integer);

procedure Wait
  (sem: semaphore);

procedure Signal
  (sem: semaphore);
```

where *sem* is the semaphore and *initval* is the initial value for the semaphore.

The synchronization between processes, such as waiting for a condition on a shared variable, is done by using the concept of an event, introduced by Brinch Hansen [1973]. There are three operations on events:

```
procedure InitEvent
  (var e: event;
   sem: semaphore);

procedure Await
  (e: event);

procedure Cause
  (e: event);
```

where *e* is the event variable, and *sem* is the associated semaphore for mutual exclusion.

An event must be initialized by a call to **InitEvent**, which associates the event with the semaphore for mutual exclusion. A call of **Await** delays the process and makes an implicit signal to the associated semaphore. A call to **Cause** by another process moves all delayed processes to the queue of the associated semaphore.

Clock handling

The procedure **WaitTime** makes the calling process wait a specified time interval, which is expressed in ticks. A tick is 20 milliseconds.

```
procedure WaitTime
  (time: integer);
```

Interrupt handling

The procedure **WaitIO** makes the calling process wait for a specified interrupt. The procedure sets the enable bit in the specified status register before suspending the running process. Some other process is then scheduled for execution. When the interrupt occurs, the enable bit is reset.

```
procedure WaitIO
  (vecaddr: integer; var statusreg: integer);
```

The process waiting for the interrupt is indicated as ready for execution and the process having the highest priority is resumed. Only one process at a time can wait for a certain interrupt and this must be guaranteed by the user.

The PDP-11 has memory mapped I/O. Pascal–1 allows manipulation of the device buffers and status registers as ordinary variables by allowing specification of addresses in the variable declaration.

Program organization

Since the programmer has to ensure mutual exclusion himself, it is important to organize the program in a way that aids in using the semaphores correctly. A natural solution is to collect the data, the semaphore for mutual exclusion, and the event variables in a record. Operations on the data are then conveniently done via a **with** statement. Ordinary Pascal allows recursion, thus procedures are reentrant, so it is possible to construct a set of procedures that operate on the shared data. This is the idea behind the monitor concept in Concurrent Pascal. Pascal–1 allows **type**, **variable**, and **procedure** declarations to be mixed, making it possible to collect the record declaration and the procedures together.

Implementation

The introduction of concurrent processes means that code, processor and storage are shared resources. The problem of handling and protecting these resources must also be considered. Our aim is to use standard Pascal as far as possible. However, as hardware facilities such as registers must be manipulated, it is not possible to make the code completely portable. These computer-dependent parts are isolated in a small set of assembly procedures called the nucleus.

Hardware

The PDP-11 computer has eight 16-bit general registers **R0** through **R7** [Digital Equipment Corporation, 1976]. The register **R6** normally serves as the stack pointer (**SP**), pointing to the top element. The stack grows downward in the memory. Register **R7** is the program counter (**PC**) of the processor.

Pascal–1 run-time organization

Figure 1 shows the memory usage for a Pascal–1 program. All global variables are indexed relative to register **R5**. The

top-of-heap pointer is a global assembly variable called **$KORE**. The stack and the heap grow toward each other. When a procedure is called or an allocation is made on the heap, a run-time check for overflow is performed.

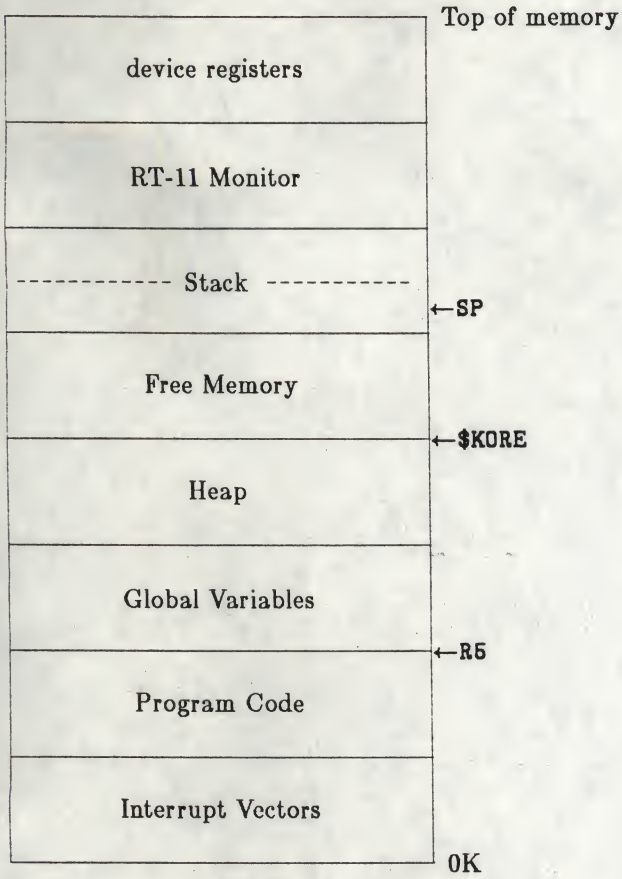


Figure 1. Pascal–1 program memory organization

When a procedure is called in a Pascal–1 program, the parameters are calculated and pushed onto the stack. The first parameter is stacked first. When all parameters are stacked, a jump to the procedure is done with the assembler instruction **JSR PC, SUBR**, which pushes the return address (**PC**) onto the stack. The procedure then allocates space on the stack for the local variables.

All local variables and the parameters are accessed by indexing with **SP**. Intermediate-level variables are accessed by the use of the static links. The static link for an invocation of a procedure is a pointer to the stack frame of the latest invocation of the lexically enclosing procedure. To access an intermediate variable, it is necessary to follow the static links until the proper stack frame is reached and index by that base value. Information on the new static link is kept in **R4** during the subroutine jump. The static link is pushed onto the stack on top of the local variables (see Figure 2). The static link is not used in global procedures.

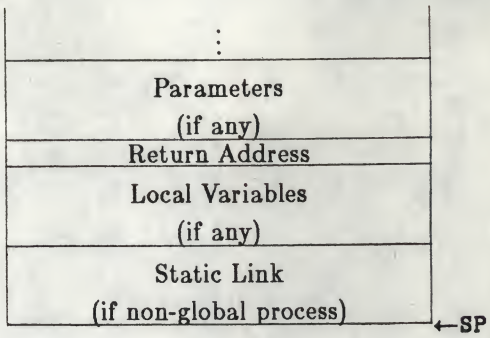


Figure 2. The stack frame

Memory and process handling

A basic requirement for concurrent processes is that they have separate stacks for local variables and stack frames of called procedures.

The main program is converted to a process by a call of the nucleus procedure **InitNucleus**.

```
procedure InitNucleus
  (memreq: unsignedinteger;
   VAR startfree, endfree: unsignedinteger;
   VAR main: process);
```

where *memreq* is the memory requirement (in bytes) for main process, *startfree* is the returned start address for free memory, *endfree* is the returned end address for free memory, and *main* is the process variable for the main process.

An estimate of the memory requirements for stack and heap of the main process must be given as *memreq*. Global variables are excluded but space for stack frames of procedures called by the main process should be included. The procedure also returns information about free memory. The information is used later when subprocesses are allocated memory.

A new process is created by calling **NewProcess**, the nucleus procedure, as follows:

```
procedure NewProcess
  (procedure proced;
   startarea, endarea: unsignedinteger;
   var proc: process);
```

where *proced* is the process-procedure for the process, *startarea* is the start address for stack and heap area, *endarea* is the end address for stack and heap area, and *proc* is the process variable for the process.

Figure 3 shows the assembly code for **NewProcess**.

```
;  procedure newprocess(procedure proced;
;     startarea, endarea:
;     unsignedinteger;
;     var proc: process);
;  Creates a process from the procedure proced
;  and associates it with 'proc'.

NEWPROCESS:
  MFPS  -(SP)            ; push(PSW)
  MTPS  #340             ; disable

  MOV   6(SP), R0        ; R0:=endarea {child.SP}
  MOV   8.(SP), R1       ; R1:=startarea
  MOV   R1, @4(SP)       ; proc:=R1
  MOV   12. (SP), R4     ; R4:=proced.staticlink

  MOV   10.(SP), -(R0)   ; pushchild(proced.startaddr);
  MOV   R4, -(R0)        ; pushchild(R4)
  MOV   R5, -(R0)        ; pushchild(R5)
  MOV   R1, R2           ; R2:=R1
  ADD   #2, R2           ; R2:=R2+2
  MOV   R2, -(R0)        ; pushchild(R2) {$KORE}
  MOV   #RESCHILD,-(R0)  ; pushchild(RESCHILD)
  MOV   R0, (R1)         ; proc^:=R0 {child.SP}

  MTPS  (SP)+            ; pop(PSW)
  MOV   (SP), 10.(SP)
  ADD   #10., SP
  RTS   PC

RESCHILD:                ; {Start of child}
  MOV   (SP)+, $KORE     ; pop($KORE)
  MOV   (SP)+, R5        ; pop(R5)
  MOV   (SP)+, R5        ; pop(R4)
  MTPS  #0               ; enable
  JSR   PC, @(SP)+       ; proced
                         ; {shouldn't come here}
LOOP:
  JMP   LOOP             ; while true do;
```

Figure 3. Procedure NewProcess

Processes communicate by using variables that are accessible for the process-procedures according to the ordinary Pascal scope rules. There is no problem using global variables since they are all accessed relative to register **R5**. The parameter *proced* contains information about the static link and the address of the code for the procedure.

A process is suspended by a call to the nucleus procedure **Resume** or **IOresume** or by an interrupt. The context (relevant registers, **$KORE**) of the process is stored on the stack (see below) of the process. The stack pointer is saved just below the heap of the process. The address to this location is the value of the process variable. The memory area of a suspended process is shown in Figure 4. The nucleus has a copy of the process variable of the currently running process in an assembly variable named **CURRENT**.

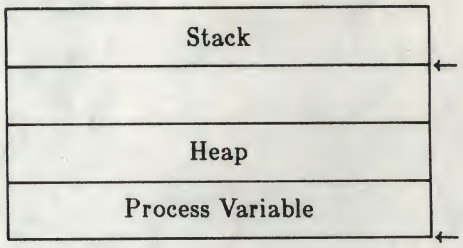


Figure 4. Memory area of a suspended process

The execution of a process **p** is resumed by a call to the nucleus procedure **Resume**:

```
procedure Resume
  (proc: process);
```

where *proc* is process variable of the process to be resumed. Figure 5 shows the assembly code activated by the call.

```
;  procedure resume(proc: process);
;  Resumes the process 'proc'.

RESUME:
  MFPS  -(SP)            ; push(PSW)
  MTPS  #340             ; disable
                         ; {Store context}
  MOV   R5, -(SP )       ; push(R5)
  MOV   $KORE, -(SP)     ; push($KORE)
  MOV   #RES, -(SP)      ; push(RES)
  MOV   SP, @CURRENT     ; CURRENT^ := SP
                         ; {Resume 'proc'}
  MOV   10.(SP), CURRENT ; CURRENT:=proc
  MOV   @CURRENT, SP     ; CURRENT:=proc
  JMP   @(SP)+           ;  switch

RES:                     ; {Restore context}
  MOV   (SP)+, $KORE     ; pop($KORE)
  MOV   (SP)+, R5        ; pop(R5)
  MTPS  (SP)+            ; pop(PSW)

  MOV   (SP), 2(SP)
  ADD   #2, SP
  RTS   PC
```

Figure 5. Procedure Resume

The nucleus procedure **IOresume** makes the current process wait for a specified interrupt and resumes another process:

```
procedure IOresume
  (proc: process;
   vecaddr: integer);
```

where *proc* is the process variable of the process to be resumed, and *vecaddr* is the vector address for awaited interrupt. When the interrupt occurs, the current process, which might not be *proc*, is suspended and the process waiting for the interrupt is resumed.

The LSI-11 has only two priority levels: interrupts enabled or disabled. The nucleus contains two procedures that change the status of the processor.

```
procedure enableinterrupts;

procedure disableinterrupts;
```

All processes, including the main program, start with the interrupts enabled.

Upon interrupt, the content of PC is automatically pushed onto the stack. PC is loaded from a preassigned memory location called an *interrupt vector*. The actual location is chosen by the device interface designer and is located in low memory address. When the specified interrupt occurs, the suspended process should be resumed. The desired effect of the interrupt is thus that a call would be done to a procedure:

```
IntResume(iosuspended: process);
```

Procedure **IntResume** is similar to **Resume**. The argument **iosuspended** is the process that called **IOresume** with the vector address corresponding to the interrupt. However, the hardware handling of the interrupts on a PDP-11 allows only the calling of a parameterless procedure when the interrupt occurs. This hardware facility is not convenient in connection with reentrant routines that are called by several processes.

One solution to this problem is to dynamically create, for each process, a small procedure that calls **IntResume** with the appropriate argument. In fact, the only thing that needs to be done is to place the assember instruction **JSR PC,INTRESUME** above the top of the stack of the process calling **IOresume** and the address to this instruction in location **vecaddr**. When the interrupt occurs and the **JSR** instruction is executed, the return address is stacked on the stack of the currently executing processes. However, the return address is exactly the stack pointer (i.e. the value of the process variable) for the I/O-suspended process. When the interrupt occurs, the process that should be resumed is made known to the interrupt handling routine in an efficient manner.

Processor management

The kernel primitives contain switches of the processor between the processes. The nucleus has procedures that can suspend and resume processes, but the kernel must decide which processes should be running. The kernel stores relevant information about the processes in records of type **processrec** (see Figure 6). The kernel keeps these records in different double-linked lists. Every list has a head of the same type as the rest of the elements in the list in order to avoid special handling of empty lists.

The processes that are competing for the processor are kept in **readyqueue**, and the variable **running** keeps track of the running process.

One list of process records is associated with each semaphore **waiting** and each event **delayed**. All processes waiting a specified time are kept in a single list **timequeue**. They are ordered according to the increasing waiting times. The process record contains one field **time**, which contains the waiting time relative to the preceding process in the time queue. The waiting time for the first process is relative to the current time. This is an efficient way of organizing the time queue because the clock process needs only to decrement the time field of the first process at each clock tick. The relative waiting times are calculated by the procedure **WaitTime**.

Only one process is allowed to wait on a specific interrupt. Therefore, no list of process records is needed. A reference to the process record of the waiting process is kept in a local variable in each instance of the procedure **WaitIO**.

The scheduling rules make it natural to order the elements in the ready queue and in the queues associated with semaphores according to their priorities. The order is unimportant in a queue associated with an event, so its elements can be inserted at the end.

Kernel structure

The data structure of the kernel is a shared resource and mutual exclusion is accomplished at this level by disabling interrupts.

The entry routines of the kernel must be visible from the user's program. The Pascal-1 compiler allows separately compiled modules with procedures and functions. A global routine in a separately compiled module is visible from other modules, if the compiler's external module switch is turned on (**{$E+}**). The user who wants to use a separately compiled routine must define a procedure heading followed by the keyword **external**. A file that contains predefined types and procedures declared external is available as a prefix to the user programs.

Conclusion

These techniques allow programmers to introduce concurrency in ordinary Pascal without changing the compiler or the support library, an approach suitable for educational purposes. The kernel is used in an undergraduate course in real-time programming. Programs for control of a physical process, operator communication and point-to-point protocol for computer communication have been implemented by the students. It is also possible to use the nucleus to test new primitives for concurrent programming. Routines for message passing and for the rendezvous concept of Ada have been implemented.

The resulting kernel is comparable to Concurrent Pascal or to Texas Instruments μP (Microprocessor Pascal). Hoppe [1980] presents a similar kernel with message-passing mechanisms written in Modula-2 [Wirth, 1980]. Some extensions have been made to the kernel in order to make it easier to work with. We've used a D/A converter, which outputs an analog signal that is displayed on an oscilloscope, to show what process is currently running. It is also possible to take a snapshot and generate a status report for the processes. A multi-terminal handler has been written in Pascal. It has been connected to the **read/write** statements of Pascal by handling the software interrupts generated by the support library.

The nucleus (Figure 6) shows what has to be added in order to obtain concurrency in Pascal. The code is written in a straightforward way to make it easy to understand. The primitives introduced in the nucleus are similar to those of Modula-2. Interrupt handling is included in an efficient way.

References

P. Brinch Hansen, *Operating System Principles* (Englewood Cliffs, New Jersey: Prentice Hall Inc., 1973).

P. Brinch Hansen, *The Architecture of Concurrent Programs* (Englewood Cliffs, New Jersey: Prentice Hall Inc., 1978).

Microcomputer Handbook (USA: Digital Equipment Corporation, 1976).

H. Elmqvist and S. E. Mattsson, "Implementation of Basic Primitives for Concurrent Programming in Pascal," (Lund, Sweden: Department of Automatic Control, Lund Institute of Technology, CODEN:LUTFD2/(TFRT-7230)/1-023/, 1981[a]).

H. Elmqvist and S. E. Mattsson, *A Real-Time Kernel for Pascal* (Lund, Sweden: Department of Automatic Control, Lund Institute of Technology, CODEN:LUTFD2/(TFRT-7231)/1-023/, 1981[b]).

J. Hoppe, "A Simple Nucleus Written in Modula-2: A Case Study," *Software Practice and Experience* 10 (1980): 697-706.

J. Kriz and H. Sandmayr "Extension of Pascal By Coroutines and Its Application To Quasi-Parallel Programming And Simulation," *Software Practice and Experience* 10 (1980): 773-789.

Pascal-1, Version 1.1 for RT-11 (Portland, Oregon: Oregon Software Inc., 1978).

N. Wirth, *Modula-2* (Zurich, Switzerland: Institut fur Informatik, ETH, 1980).

Professors Elmqvist and Mattsson teach real-time programming in the Department of Automatic Control at the Lund Institute of Technology in Sweden. They have designed a kernel that supports concurrent programming in Pascal-1. Their approach shows what must be added to set concurrency in a sequential language and it is therefore suitable for education. This description of their real-time kernel was presented at the IFAC Symposium on Software for Computer Control 1982 and is reprinted here by permission of the International Federation of Automatic Control. The authors want to thank Leif Andersson for many good ideas and stimulating discussions.

```
{Real-Time Kernel for Pascal.}
{Use the nucleus prefix.}

CONST maxpriority = 1000;

TYPE unsignedinteger = 0..65535;
     processref =^processrec;
     semaphore =^semaphorerec;
     event =^eventrec;
     processrec = RECORD
     succ, pred: processref;
     proc: process;
     priority: integer;
     time: integer;
     END;
     semaphorerec = RECORD
     counter: integer;
     waiting: processref;
     END;
     eventrec = RECORD
     reentry: semaphore;
     delayed: processref;
     END;
VAR  running, readyqueue, timequeue: processref;
     freetop, freebase: unsignedinteger;

PROCEDURE put(p, q: processref);
{Inserts process record p before process record q in q's list.}
  BEGIN
  p^.succ:=q;
  p^.pred:=q^.pred;
  q^.pred^.succ:=p;
  q^.pred:=p
  END;

PROCEDURE remove(p: processref);
{Removes processrecord p from its list.}
  BEGIN
  WITH p^DO
    BEGIN
    pred^.succ := succ;
    succ^.pred := pred
    END
  END;

PROCEDURE putpriority(p, q: processref);
{Inserts processrecord p in queue q according to priority.}
  VAR p1: processref; pri: integer;
  BEGIN
  pri := p^.priority;
  p1 := q^.succ;
  WHILE (p1 <> q) AND (pri )= p1^.priority)
    DO p1:=p1^.succ;
  PUT (p,p1)
  END;
```

```
PROCEDURE schedule;
  BEGIN
  IF readyqueue^.succ <> running THEN
    BEGIN
    running := readyqueue^.succ;
    resume(running^.proc);
    END;
  END;

{$E+} {external}
PROCEDURE createprocess(PROCEDURE proced; memreq: unsignedinteger);
  VAR child: processref;
  BEGIN
  disableinterrupts;
  freetop := freetop - memreq;
  new(child);
  child^.priority := 1;{Default priority.}
  putpriority(child, readyqueue);
  newprocess(proced, freetop, freetop+memreq, child^.proc);
  schedule;
  enableinterrupts;
  END;

PROCEDURE initkernel(memreq: unsignedinteger);
  CONST clockarea = 100; idlearea = 100;
  BEGIN
  disableinterrupts;
  {Create readyqueue with running.}
  new(running);
  new(readyqueue);
  readyqueue^.succ := running;
  readyqueue^.pred := running;
  running^.succ := readyqueue;
  running^.pred := readyqueue;

  {Create empty time queue.}
  new(timequeue);
  timequeue^.succ := timequeue;
  timequeue^.pred := timequeue;

  running^.priority := 1;
  initnucleus(memreq, freebase, freetop, running^.proc);

  createprocess(clock, clockarea);
  createprocess(idleproc, idlearea);
  END;
```

```
PROCEDURE initsem(var sem: semaphore; initval: integer);
  BEGIN
   new (sem);
   WITH sem^ DO
     BEGIN
     counter:=initval;
     new(waiting); {Empty waiting queue.}
     waiting.succ:=waiting;
     waiting.pred:=waiting;
     END;
  END;

PROCEDURE wait(sem: semaphore);
  BEGIN
  disableinterrupts;
  WITH sem^ DO
    BEGIN
    IF counter >0 THEN
      counter := counter -1
    ELSE
      BEGIN
      remove(running);
      putpriority(running, waiting);
      schedule
      END;
    END;
  enableinterrupts;
  END;

PROCEDURE signal(sem: semaphore);
  VAR p:processref;
  BEGIN
  disableinterrupts;
  WITH sem DO
    BEGIN
    IF waiting <> waiting^.succ THEN
      BEGIN
      p := waiting^.succ;
      remove(p);
      putpriority(p, readyqueue);
      schedule
      END
    ELSE
      counter:=counter+1
    END;
  enableinterrupts;
  END;
```

Figure 6. This listing contains some parts of the kernel. The procedures **SetPriority**, **Initevent**, **Await**, **Cause**, **WaitIO** and **WaitTime** and the processes **Idleproc** and **Clock** are not included.

The Log: Pascal-1 and Pascal-2

Oregon Software's previous Pascal Newsletter described the significant changes in Pascal–2 V2.1A. This log details bugs fixed in Pascal–2 V2.1B, which may now be ordered. Several fixes listed for V2.1 involve the support library; such fixes also apply to Pascal–1.

The changes described apply to all versions of Pascal–1 or Pascal–2 unless a specific operating system is specified. When possible, work-arounds are given for V2.1A users.

Pascal–2

Version 2.1B corrects these problems for all PDP-11 systems.

Debugger altered registers

The Debugger altered some of the program's registers when the `C`, `S`, or `P` commands were used to start a program running under the Debugger. Work-around: use `G` to begin program execution. After the program has started executing, the `C`, `S`, and `P` commands work correctly.

Negative zero

The `write` procedure sometimes printed a negative sign for negative real numbers that round to zero. For example, `write(-0.010:7:1)` printed the value –0.0. The correct result is 0.0. This error did not occur with all such numbers.

'Sin' and 'Cos' problem

The double-precision `sin` and `cos` routines did not return the correct value when their argument was exactly 0.0. Work-around: use a special check for the argument 0.0, as in the following example:

```
function Xcos(X:real):real;
begin
  if X = 0.0 then Xcos := 1
  else Xcos := Cos(x)
end;
```

Line numbers thrown off

The use of included files threw off error walkback line numbers. Work-around: use the new syntax, with quotes around the file name.

```
%include 'filename';
```

The old syntax may not continue to be valid in the future, anyway.

Debugger changed variables

When a fatal error is detected in a user's program, the Debugger regains control after the error message is printed. However, the Debugger did not print the correct values for some variables, because the error-reporting procedures in the support library did not preserve registers and the Debugger cannot print the values of variables stored in those registers. V2.1B fixes the problem for many run-time errors, but certain kinds of run-time errors may still change the values of some variables. Please be aware of this restriction: **When a run-time error terminates a program, some variable values printed by the Debugger may be inaccurate.**

Dynamic allocation incorrect

When a 2-byte block was allocated from a 4-byte free block in memory, the remaining 2 bytes in the free block were not handled correctly during execution. The problem caused memory fragmentation or even odd address traps when the program ran.

Multiple errors on file output

The error message mechanism attempts to write any partial output line to the output file as part of the `close` operation after a fatal error. If the error was detected during a write to an output file, such as a disk, the attempt triggered the "multiple errors detected" message instead of the actual problem. In V2.1B, the support library marks the second occurrence of an I/O error for file output as "non-fatal." This should prevent the program from aborting with the "multiple errors detected" message.

Miscellaneous

Incorrect results were obtained from functions that returned structured-type values.

Run-time traps and incorrect calculations occurred when operations were performed on packed structures.

Spurious compile-time errors were reported when conformant array parameters were used with nested procedures.

Changes to RSX

Version 2.1B for RSX corrected these problems:

'Getpos' returned incorrect location

The `getpos()` procedure did not always return the correct location of the next record about to be read when the input file was a `text` file.

'Put' didn't always put

When a terminal, line printer, or paper tape punch was opened as a file type other than `text`, data was not correctly written to the device when a `put()` was used. For

example, if a terminal was opened as a **file of integer**, and **put()** statements used to write data to the terminal (two characters per integer), nothing was printed on the terminal.

'Ioerror' always 'true'

If **ioerror** was the first operation performed on a **text** file, an interaction with Lazy I/O caused the return of the value **true** even when no error existed. Work-around: use the fourth parameter of the **reset** or **rewrite** call to determine whether the file is correctly opened. The value of the fourth parameter is -1 if the file cannot be opened.

Errors not trapped

The run-time errors "File is not a random access file" and "Seek() to record zero" were not correctly trapped when **noioerror()** was used with a file.

Profiler overlay omitted

The overlay description file PAS.ODL for V2.1A did not explain how to overlay a Pascal–2 program compiled with the /PROFILE option. For V2.1B, PAS.ODL properly describes overlay procedures for programs that use the Profiler.

'Not enough memory'

When a Pascal task is initialized and the section **$$HEAP** has not been expanded, the support library attempts to allocate 2K words for the stack by extending the task. This works fine until the size of the task reaches 30K words. At this point, less than 2K words remain for the stack and the compiler reports "not enough memory." In 2.1B, Pascal–2 uses any remaining space for the stack when a task exceeds 30K words, allowing users to write larger tasks before using overlays.

Incorrect file size returned

When the fourth parameter is used with the **reset** procedure, Pascal–2 should return the size of the file, in blocks. However, Pascal–2 was returning a file size of one block for an empty file. The file size for an empty file is now reported as zero.

/APD/RW causes lost line

A line of information may be lost under the following conditions: an existing **text** file is opened via a **reset** with the /APD/RW switches to permit appending data to the file, and a **write** statement is used without a **writeln** to write the characters to the file, and the file is then closed. The problem is caused by the use of **reset** to open the file; **reset** normally opens files for input only. When the /APD/RW switches are used, the file is really an output file, so any partial lines should be flushed out when the file is closed. Use **writeln** to be sure that the last line is written to the file before the file is closed.

'Sayerr' incorrect

The **sayerr** routine did not correctly print the text for I/O error when the RSX I/O error codes and directive error codes (stored in a byte) were ambiguous. In version 2.1B, the directive error codes are biased by 128 to distinguish them from I/O error codes. V2.1A users should be aware that errors such as "Illegal user buffer" might be reported when "Device driver not resident" is actually the error.

Record lengths for 'text' files

When **text** files containing lines longer than 132 characters were opened with a **reset** statement, a "record length error" could be generated because Pascal allocates a maximum buffer size of 132 characters. Work-around: use the I/O control switch **/VAR:***nnn* where *nnn* is the maximum line length of the data in the file. The value *nnn* can be greater than 132.

'Rename' didn't

When an explicit version number was given for a file to be renamed and that file was not the most current version, the **rename** operation completed as expected, but the directory entry for the most current version of the file was removed instead of the entry for the file that was renamed. V2.1A users should not give explicit version numbers for files being renamed.

Character dropped on line wraps

When a line longer than 132 characters is written to a **text** file, the line wraps and the line is broken at 132 characters as if by a **writeln** statement. Under V2.1A, a character was dropped when the line is wrapped this way on disk files. Terminal output wraps correctly. Work-around: use a **writeln** statement before outputting more than 132 characters.

Message vague

The run-time error message "multiple errors detected" does not provide any information about the actual errors detected in the program. The error message is printed when a run-time error occurs while an error message is being printed, and it usually indicates a severe error, such as writing over support library code. In version 2.1B, the support library causes a run-time trap so that RSX prints the contents of the hardware registers as follows:

R0	User PC of original error
R1	Error code for original error
R2	Secondary error PC
R3	Secondary error code
R4	I/O status if I/O error

The error codes are in Appendix B: Run-Time Error Messages of the Programmer's Guide.

Errors, additions to manuals

Sample program errs

The sample program REVERSE listed in the *Pascal-2 User Manual* for V2.1 does not operate properly. The second line of the program being reversed is not printed because of an interaction with lazy I/O and the way the REVERSE program is written. To operate correctly, a boolean variable called **Done** must be inserted into the program. The main loop in the program must be modified as shown:

```
repeat
  new(x);
  with x^ do Getpos(f, block, offset);
  x^.next := p;
  p := x;
  Done := eof(f);
  if not Done then readln(f);
until Done;
```

Changes to RSTS

Version 2.1B for RSTS corrects these problems:

No record-oriented I/O under RSX

When the RSX version of Pascal is run under the RSX emulator on RSTS, it is not possible to perform I/O on record-oriented devices other than the user's terminal. The RSTS system returns an error status of −37 for all such I/O requests because the operating system does not properly simulate RSX I/O to record devices such as the line printer. I/O to the user's terminal is handled correctly. In V2.1B, code has been added to enable the support library to recognize RSTS emulation and perform I/O to record devices correctly.

'Read' may hang

A **read** performed on an integer or real hangs forever if the user types ^Z with I/O error trapping turned on and without entering a valid number first.

Second real value lost

The value of the second real variable is lost in programs that attempt to read two reals in succession, but only when the second real contains fewer characters than the first. Workaround: add a leading 0 to the second real.

Iostatus printed incorrect values

Iostatus produced the wrong values when an I/O error occurred, causing the error routine to print either a zero or a large negative number.

Modules missing in /eis version

The EIS version of the support library for V2.1A did not contain modules for the exponent function and the natural logarithm.

Miscellaneous

LIBDEF.PAS was not properly reflecting the data stored in the file variable. This resulted in FDUMP not giving proper results as well as prohibiting the user from testing any of the device status conditions accurately.

OPFDMP.PAS has been modified to properly print the device number if there is one.

OPERRO.PAS now prints the unit number of the device if an I/O error occurred.

The entry point for **P$DISP** was in the user library when it should have been in the run-time system. Attempts to run the examples in the manual that accessed this entry would fail.

Changes to RT-11

Error reporting improved

The **fatal** error status at location 53 octal was not being properly set or tested by the support library. This resulted in some command files not terminating when a fatal error was encountered.

In conjunction with the error status problem, the low-level routines (.READ/WRITE) would also set the error condition bit if a fatal transfer error occurred. This would cause Pascal to print the "multiple errors" message.

The **Sayerr** routine was added to print the error text of low-level RT-11 system calls.

Iostatus would produce the wrong values when an I/O error occurred, causing the error routine to print either a zero or a large negative number. **Iostatus** has been modified to return a negative value if a system error occurs on a file, a zero if no error occurs, and a positive number if the problem is a support library error. The error number is returned as a full 16-bit integer value.

The support library now saves the error condition code (if any) from the last **reset** or **rewrite** operation so that the user may call **Ioerror** and **Iostatus** to determine a possible cause if the operation fails. This will work properly only if called before any other I/O operation is performed.

The reason is that the saved location is also set by the .READ/WRITE code since **reset** could imply a .READ, which could cause the failure.

'Read' may hang on reals

A **read** performed on an integer or real hangs forever if the user types ^Z with I/O error trapping turned on and without entering a valid number first.

Second real value lost

The value of the second real variable is lost in programs that attempt to read two reals in succession, but only when the second real contains fewer characters than the first. Workaround: add a leading 0 to the second real.

Pascal has a standard

Pascal now officially has an international standard. The International Standards Organization, in a vote tallied this summer, adopted a standard language definition for Pascal. The action followed earlier adoption of an American standard that is a subset of the international one, lacking only conformant array parameters.

The sequence leading to adoption of the Pascal standard began in December 1982, when ANSI and IEEE agreed on the American standard, identical to the international draft standard except for conformant array parameters. At the same time, the Joint Pascal Committee of ANSI and IEEE recommended adoption of the international standard Level 1 (including conformant array parameters) to the U.S. committee known as X3J9, which then voted "yes" on the international standard at the next meeting. Previously, the U.S. was one of three "no" votes. This time, the ISO standard passed with no dissenting votes and one abstention.

Work continues on extensions

The Joint Pascal Committee of ANSI and IEEE continues to work on a "candidate extension library" for extensions that will resolve the known weaknesses of standard Pascal. The ANSI-IEEE committee has agreed on a number of minor issues but has not resolved differences over major issues. Proposals are circulating on these issues, which remain "work in progress." A report is due in December.

Copies of the international standard are available from:

Document number BS 6192:1982.
British Standards Institute
Marylands Avenue
Hemel Hempstead
Herts HP 2 4SQ

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the "Information Exchange." Your description should follow the format of the items below. Interested parties may contact one another directly.

Log Management System and RMS Interface, for RSX-11M; a menu-driven 50-program package that incorporates a custom-developed interface (available separately) between Digital's Record Management Services (RMS) and Pascal–2 V2.OK. The log package supports a multiplicity of data reporting features with subsystems for accounting, timber sales, and sales data accumulation. The RMS interface gives the user access to all of the RMS sequential, random, and multi-keyed capabilities. Contact: Jim Bombardier, Wasser and Winters Co., P.O. Box 396, Longview, WA 98632, (206) 423-1080.

DCL Command PASCAL, for the Oregon Software Pascal–2 V2.1A compiler; software package contains patches and files to enable a new DCL command **PASCAL** and to make **HELP PASCAL** work. All files are for RSX-11M V4.0 only. For information, contact: Mr. Bruce Williams, OPUS Coordinator, c/o EOCOM, 15771 Red Hill Avenue, Tustin, California 92680.

Distributors attend annual workshop

Oregon Software Distributors from Europe, Canada, Japan and Australia received technical information from Oregon Software development teams and exchanged viewpoints on the needs of customers at their second annual workshop.

The two days of discussions in Portland followed DEXPO East and preceded Oregon Software's sixth Annual Open House. The first day was devoted to technical presentations and a hands-on session, where distributors ran concurrent-programmed games on the 68000, used SourceTools in a simulated development environment, and compiled programs under UNIX on the PDP-11. The second day was devoted to round-table discussions in which participants exchanged views on the needs of customers, the quality and timeliness of written materials, and the need for additional information.

Customer contact is the key for sales staff

In the last issue, we introduced our marketing director, David Cloutier, and manager for general distributors, Pat Rau. We'd also like you to know our sales staff, who they are and what they do.

Lorie, Terry, and Tim handle customer support and sales. Their jobs require them to spend lots of time on the phone, providing product information, following up on sales, and responding to customers' requests for help. They respond to inquiries from prospective customers, which come in response to advertising, articles, or as referrals from our friends. The sales staff provides prospects and current customers alike with additional product information.

Their jobs don't end with a sale. They rectify any shipping problems and keep their accounts in support.

Lorie Griffith

Lorie, our senior sales person, has a strong science background, including some contact with computers as a chemistry major in college and as an assayist on her first job. Later, she became a computer operator for a sawblade manufacturer. Before she joined Oregon Software, Lorie did customer support for an applications software company. Her major project was a large automated inventory and accounting system for an automobile importer. As part of an 18-month contract, she did some programming and wrote user documentation in addition to providing support.

Lorie lives in rural Washington, where she raises livestock and is active in community projects.

Tim McMenamin

Tim says that "people make the job worthwhile"; he likes the service to customers as much as the software itself. He enjoys negotiating with distributors and finds the contact with large firms stimulating.

A native of Wisconsin, Tim came to Oregon for skiing and schooling. A journalism and political science major at college, he worked on the staff of a state senator and a congressman. Before coming to Oregon Software, Tim sold DBMS applications and CP/M program generators for a small software house. He likes the technical challenge of sales and is taking computer science courses to build his programming knowledge.

When he's not at work or school, Tim enjoys all forms of skiing, camps out in the local mountains, and runs for exercise.

Terry Juve

Terry is a recent addition to the staff. Trained as a programmer, she likes to work with other programmers. She also enjoys direct contact with the customer.

Terry recently earned an associate degree in applications programming at Portland Community College, and she's starting on a bachelor's degree in computer science at Portland State University.

Her sales background has been in commercial casuality insurance for various brokerage houses in Portland, and she's taken business courses as well.

Between work and school, she likes to spend the little bit of free time she has reading science fiction and psychology; she likes to cook, hike, dance, and play an occasional softball game.

Bug Contest winners crowned

After some tough decisions, we have the winners for the Bug Contest (see Newsletter No. 4). We couldn't decide on a "best" bug so we have co-winners. The prize category itself forced us to make some hard choices, but we finally decided on an official Oregon Software "Party Pascal" baseball hat, plus a bottle of Oregon wine with which the winners may begin a party at which they may wear their new hats. The winners are:

Best Bug	Herje Wikegaard (SERN Dator Konsult) Lee Mattiesen (NW Oklahoma State U.)
Application	Joann H. Schultz (Lockheed Electronics)
Best Prize	Doug Foster (E-systems)

This issue marks a transition

Oregon Software's seventh Pascal Newsletter marks a transition. David Spencer officially takes over as editor, and I step down. Actually, David has coordinated the writing and production of the last several issues. I have written a few articles and have assisted with the review and rewriting of others, but David has been doing the vast majority of the work.

David is also taking over as manager of the technical writing group. I am returning to the ranks of simply "writer." This changeover seems to be an appropriate time to review our company's writing efforts since July of 1980, when Oregon Software's first technical writer (yours truly) was hired.

We instituted the newsletter with the ultimate goal of quarterly issues. The one-man writing staff started slowly. After adding two staff members — David and Mike Kuhn, who has been our primary manual specialist — we have quickened the pace for a total of 7 in about 30 months. Time constraints still force us to combine issues on occasion, but the result has been the publication of newsletters with more substance. This one, for instance, is the combined Summer-Fall issue, with special emphasis on concurrent programming. It's the largest and most technically oriented issue this year.

We plan to continue with at least one staff-produced technical article and one user-produced article each issue. You should note that we run as many good user articles as we get, and we're always looking for more. Please write or call David with your ideas. We'll also continue with our regular fare of product announcements, OPUS columns, book reviews, letters, etc. We hope to have a more detailed Bug Log, including better lists of known bugs, both fixed and unfixed. Our biggest problem is that the production and mailing processes require several weeks' time, and the number of known bugs, and their status, can change dramatically after the newsletter has gone to press.

For our documentation in general, our goal has been to have manuals that reflect the high quality of our software. Further, we wanted a plain style that described complex software in the simplest terms possible. To a large degree, I believe we have succeeded. This summer marked the release of our new 2.1 manuals. Mike was the primary author of the new manuals, which not only document new software features but also provide much more detail on existing features. The considerable expansion of the 2.1 manuals represents about six months of Mike's hard work, with occasional help from the rest of the writers and a good deal of assistance (as always) from our programming staff.

In addition to manuals for new products, including SourceTools and several Pascal–2-based development systems, we are also working on new manuals for our original Pascal product, Pascal–1. A supplement to the existing Pascal–1 manuals is complete, and a revised manual similar to the Pascal–2 manuals should be done this fall. One of David's biggest tasks will be that of bringing all of our manuals up to the uniformly high standards of our Pascal–2 product line.

We need your help, by the way. Many of the improvements in our manuals have resulted from comments by users — a small but vocal group of users, as it happens. I'd like to take this last opportunity as outgoing editor to solicit comments and questions about our newsletter and documentation. Though our materials are reviewed thoroughly in-house before release, we can't really know whether they meet the needs of the users unless you tell us.

Collins Hemingway

Pascal NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

David Spencer, editor
Collins Hemingway and Michael Kuhn, staff writers
Erin Ryan, production assistant

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. We pay $100 per newsletter page for any article we print.

Printed in USA

Pascal
NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

Pascal
NEWSLETTER

Number 8 OREGON SOFTWARE Spring/Summer 1984

Resident libraries save memory

by Steve Poulsen

Prior to Version 2.1B, a Pascal–2 resident library contained both Pascal support library subroutines and File Control Services I/O routines from the system library (LB:[1,1]SYSLIB.OLB). You can now build Pascal resident libraries that do not contain File Control Services (FCS) subroutines. You can also create a memory resident overlaid Pascal library which can be clustered with an FCS resident library such as FCSRES. These new options both save substantial amounts of memory.[1]

The use of cluster libraries greatly reduces task sizes and the amount of virtual memory used, increasing memory available for code and data. System swapping overhead is reduced because each task is smaller and more tasks can fit in memory. Disk space is also conserved because each task image does not contain code from the Pascal library and the system library.

To build a Pascal resident library, first install your Pascal system with the PASBLD command file. Be sure that the Pascal support library LB:[1,1]PASLIB.OLB is the current version. Then use the command file PASRES.CMD. It asks you whether you want to build a memory resident overlaid version of the Pascal support library. If your system supports virtual (PLAS) overlays, select this option. The Pascal support library is built as two overlays. The total physical memory required for the library varies between 6K and 8K words, depending on your hardware configuration. The advantage of virtual overlays is that the library is mapped into your task using only one Active Page Register (APR). This means that the 6K to 8K words of library code can be accessed using only 4K words of virtual address space, for a savings of 2K to 4K words.

If you do not wish to use virtual overlays, PASRES.CMD can build a non-overlaid resident library, which requires 4K words of memory. This may be necessary on some RSX systems, such as VAX/VMS in compatibility mode, which do not support virtual overlays.

After PASRES.CMD builds the library and its associated symbol table, you need to create a partition called PASRES and install the resident library in that partition. (You can use the program VMR to make the partition permanent.)

Before you build the partition, you must examine the map (PASRES.MAP) to determine the total size of the library. See the portion of a sample PASRES map that follows for an example of how to do this.

The task size is given (in decimal words) in the summary at the top of the map. Convert this value to octal bytes and round up to the next multiple of 100(8): 6464. words works out to 31200 octal bytes. The last two zeroes are dropped. Base is the base address where the partition will be loaded.

Note that even though the task size is 6464. words, the task address limits are only 160000 through 177777, or 4K words. The address limits are reduced because virtual overlays are being used. This is reflected in the overlay description which shows that the segments PASIO and PASFLT overlay each other. The savings realized by using virtual overlays is 4878 bytes in this case.

Sample PASRES Map

```
Partition name : PASRES
Identification : 2.1B
Task  UIC      : [2,16]
Task attributes: -HD
Total address windows: 1.
Task  image  size  : 6464. words
Task address limits: 160000 177777
R-W disk blk limits: 000003 000034 000032 00026.
```

[1]Refer to the Pascal–2 User's Manual, Version 2.1B for RSX-11.

In this issue...

Resident libraries . . . . . Page 1
Information Exchange . . . . . Page 3
New Syntax checker . . . . . Page 4
I/O devices . . . . . Page 7
I/O switches . . . . . Page 11
New people and positions . . . . . Page 13
The Log . . . . . Page 15
A Case Study . . . . . Page 17

```
PASRES.TSK;1 Overlay description:

Base    Top         Length
----    ---         ------
160000  157777  000000  00000.     PASROT
160000  177411  017412  07946.          PASIO
160000  171415  011416  04878.          PASFLT
```

Now that you know the task size, you can create the partition, such as shown in Example A, a sample partition allocation for a small RSX system. The first column of numbers after the partition name is the location of the Partition Control Block within the monitor and can be ignored. The second column of numbers is the base address of the partition in memory. The third column is the length of the partition. With partitions, all addresses and sizes are divided by 100 (octal) to reduce the range of the numbers. The size needed for PASRES is 31200 bytes. Use the SET /TOP command to reduce the size of the GEN partition by that amount, as shown in Example B.

Example A: Original Partition Allocation

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 515700 MAIN SYS
       042620 242100 020000 SUB  (...MCR)
```

Example B: New Partition Allocation

```
>SET /TOP=GEN:-312
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 464500 MAIN SYS
       042620 242100 020000 SUB  (...MCR)
```

As you can see, the size of the GEN partition has been reduced by 31200 bytes. Compute the base of the new PASRES partition by adding the GEN partition's base and length (242100 + 464500 = 726600). Now you can create the PASRES partition based at 7266 with a size of 312.

The command SET /MAIN=PASRES:7266:312:COM creates a partition called PASRES, based at 726600 with a size of 31200, shown in Example C. It also marks the partition as a common partition. Using the command INS LB:[1,1]PASRES, you can now install PASRES.TSK into the partition so that it can be used by Pascal programs.

Example C: PASRES Partition

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300SUB  DRIVER -DL:
       066570 152200 001400 SUB DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB (F11ACP)
GEN    066314 242100 424500 MAIN SYS
       042620 242100020000 SUB  (...MCR)
PASRES 043634 726600 031200 MAIN COM
```

To build a Pascal task which uses PASRES, first compile the program in the normal way. Use the LIBR option to specify that PASRES should be used in read-only mode:

```
>TKB
TKB>TEST/FP/CP=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>LIBR=PASRES:RO
TKB>//
```

When you run the program Test, RSX automatically associates the PASRES library with your task. Your program is mapped into low virtual addresses, and PASRES is mapped into the virtual address range from 160000 through 177777.

If your RSX system supports clustered libraries (RSX-11M V4.0 or later), you can use the cluster option to specify that PASRES should be clustered with the FCS resident library FCSRES. The advantage is that the same APR can map both the overlaid PASRES as well as an overlaid FCSRES. In the partition list shown in Example C, FCSRES and PASRES are both about 6K words long, for a total of 12K

words. By clustering the two libraries together, 12K words of code can be mapped with one 4K word APR for a savings of up to 8K words in the program.

PASRES may be clustered with other libraries besides FCSRES, with the restriction that PASRES must be specified first in the CLSTR option.

When you link a Pascal task with FCSRES, you may get an error status of -39 when you try to open a file, indicating that no buffer space is available for the file. When FCSRES is not used, Pascal programs can intercept the FCS entry points which allocate and deallocate file buffers when a file is opened and closed. Pascal converts these calls to allocate memory from the Pascal heap when files are opened. When FCSRES is used, Pascal cannot intercept the FCS calls which allocate file buffers. In this case, FCS will look in the file storage region for space to allocate file buffers. The size of the file storage region should be very small since Pascal usually uses the heap for file buffers.

To avoid such errors when using FCSRES, allocate space in the program section $$FSR1 for buffers for all the files you intend to open at any one time. Each file requires about 528 (decimal) bytes.

You can allocate this space in either of two ways:

1) If you are not using PASRES but are using FCSRES, you can allocate the space in $$FSR1 by using the EXTSCT option.

2) When you are using PASRES, Pascal automatically enables the ACTFIL option of the Task Builder. The ACTFIL option permits you to specify the number of files you will have open at one time. If the value you specify with ACTFIL is greater than one, you also must use the UNITS option to make more logical unit numbers available to the task. The default value for ACTFIL is four open files.

The simple program Test demonstrates three ways to reduce memory requirements. When this program is linked in the standard way, the task size is 9376. words.

```
program Test;
  var
    f: text;
  begin
    rewrite(f, 'TI:');
    writeln(f, 'It works');
  end.
```

This file was compiled with the /NOWALKBACK switch to produce a small object module.

```
>PAS TEST/NOWALKBAC
>TKB TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
```

This program can also be linked with PASRES, reducing the size of the task to 6720. words because most of the code loaded from the Pascal support library is being shared with other tasks in PASRES. The ACTFIL option is used to set the number of open files to one.

```
>TKB
TKB>TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>ACTFIL=1
TKB>LIBR=PASRES:RO
TKB>//
```

You can save more memory by clustering PASRES with FCSRES. In this case, the size of the task is 3296. words, a savings of about 6K words over the size of the original task. When the program Test was linked without PASRES or FCSRES, the task image required 39 disk blocks. When the task is built with a clustered PASRES and FCSRES, the task image takes up only 15 blocks.

```
>TKB
TKB>TEST/FP/CP,TEST=TEST,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>ACTFIL=1
TKB>CLSTR=PASRES,FCSRES:RO
TKB>//
```

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the Information Exchange. Your description should follow the format of the items below. Interested parties can contact one another directly.

Syntax checker for Pascal–2 sources checks the lexical and syntactical structure and provides a method for producing paginated listings (see article in this issue). Available from Unit-C Limited, Dominion Way West, Broadwater, Worthing, Sussex, England.

Pascal programmer wanted to design and program a data base system for a network of microcomputers. Contact John Foley, Carghill Terminal 4, Portland OR 97203, (503) 286-1842.

Syntax checker avoids compilation bottlenecks, speeds software development effort

by Ralph Hodgson

Mr. Ralph Hodgson has worked for Eurotherm since 1976 and is now technical manager for Unit-C Limited, a subsidiary. He is a member of ACM and IEEE and is interested in real-time computing and program development tools.

Whether we are communicating with people or computers, syntax is what gives precise meaning to our words. We learn from an early age that we can be imprecise in our syntax at times, yet still be understood. The people with whom we communicate are able to fill our syntactic gaps through the context of conversation. Later we learn that a computer is not as forgiving and that with a machine, our syntax must be correct and unambiguous. Because of our previous experiences in communicating, most of us find it frustrating to be stymied by the lack of a simple semicolon or end; statement. Our frustration increases as we waste minutes of compilation time before being told that we have not been understood.

Unit-C's Syntax Checker, UPS, is a fast, efficient precompilation tool designed to reduce the frustration of discovering syntax errors. UPS quickly checks Pascal–2 source code for textual and syntactic correctness and produces a listing file without first compiling the source program.

UPS Features

UPS supports all syntax of Pascal–2's implementation, including embedded switches. Vertically arranged, numbered pointers indicate errors appearing in the source code, and all error reporting occurs in the vicinity of the error, as shown in Figure 1.

As a listing utility, UPS reports structural information for each source line in a program. Figure 2 shows the listing that UPS produces including line numbers, nesting,

Figure 1

```
 3  2          PROCEDURE NextDigit(N: integer);
 4  2            VAR J: integer;
 5  2            BEGIN
 6  2  1  1        WHILE N > Base DO
 7  2  1  1          BEGIN
 8  2  2  3          J:=N DIV Base NextDigit(J); N:=N - (J  Base))
                                  0                            0
                                  4                            5
                                  1                            3
                                  0                            2
                                  v                            v
(410) Expression incopmlete, missing operator or semicolon
(532) Unexpected ')' -- Check for matching parenthesis
 9  2  1  3                END;
10  2  1  4             CASE Base OF
11  2  1  5                 1,2,3,4,5,6,7,8,9,10: EmitCh(Chr(N + Ord('0'));
                                                                          0
                                                                          2
                                                                          1
                                                                          3
                                                                          v

(213) ')' expected
12  2  1  5             OTHERWISE IF N > 10 THEN EmitCh(Chr(N - 10 + Ord('A'))
13  2  1  8                       ELSE EmitCh(Chr(N + Ord('0')))
14  2  1  8             END Case
15  2                END NextDigit ;
```

Figure 2

```
Pascal Syntax Checker RSX11 V1.1 Site 1312-0  19-Mar-1984 21:57:10 Page 1
Unit-C Ltd Dominion Way West, Broadwater Worthing W. Sussex  BN14 8NT ENGLAND
Copyright 1984 Unit-C Ltd.
listex/list

 1                -- Demonstrates listing features of the syntax checker UPS
 2
 3               PROGRAM ListExample;
 4
 5  1               PROCEDURE ConvertNumber(Number, Base: integer;
 6  1                                          PROCEDURE EmitCh (Ch: Char));
 7  1
 8  2                 PROCEDURE NextDigit(N: integer);
 9  2                    VAR
10  2                       J: integer;
11  2                    BEGIN
12  2  1  1                 WHILE N > Base DO
13  2  1  1                    BEGIN
14  2  2  2                    J := N DIV Base;
15  2  2  3                    NextDigit(J);
16  2  2  4                    N := N - J  Base
17  2  1  4                    END;
18  2  1  5                 CASE Base OF
19  2  1  5                    1, 2, 3, 4, 5, 6, 7, 8, 9, 10:
20  2  1  6                      EmitCh(Chr(N + Ord('0')))
21  2  1  6                    OTHERWISE
22  2  1  8                      IF N > 10 THEN EmitCh(Chr(N - 10 + Ord('A')))
23  2  1  9                      ELSE EmitCh(Chr(N + Ord('0')))
24  2  1  9                    END Case
25  2                    END NextDigit ;
26  1
27  1                 BEGIN ConvertNumber
28  1  1  1              NextDigit(Number)
29  1                 END ConvertNumber;
30  1
31  1               PROCEDURE OutCh(Ch: Char);
32  1                  BEGIN
33  1  1                 BEGIN Nesting
34  1  2                  BEGIN
35  1  3  1                 write(Ch)
36  1  2  1                END
37  1  1  1               END
38  1                 END OutCh ;
39  1
40                 BEGIN ListExample
41     1  2           ConvertNumber(201,  2, OutCh); Writeln;
42     1  4           ConvertNumber(201,  8, OutCh(; Writeln;
43     1  6           ConvertNumber(201, 10, OutCh); Writeln;
44     1  8           ConvertNumber(201, 16, OutCh); Writeln
45                 END.
```

blocking and inclusion levels and statement numbers. You can use this information for checking control flow, statement blocks and lexical nesting procedures. The nesting and blocking levels are particularly useful for programmers faced with unravelling deeply nested program structures.

Running the Syntax Checker

UPS is scheduled in a manner consistent with the running of the Pascal-2 compiler [Editor's note: to "schedule" means to "invoke"]. Some examples are shown in Figure 3. Rather than a schedule command, UPS reports a summary of the manual in the form of help text. This includes a list of recognized options as given in Figure 4. You need to specify options only to lengths sufficient to make their interpretation by UPS unambiguous.

You can easily control the format of a UPS listing. In addition to the directives, **length** and **width**, for page size, controls can process included text and pagination at syntactic points in the source. You can alter the second line of the list banner to provide a unique identification to the listings.

Automatic pagination improves a Pascal listing by inserting page breaks where space is insufficient for a clean start to a new procedure. The directive clause, **break=*n***, causes page breaks at new procedures when there are less than *n* lines remaining for procedural declarations on the current page.

By default, included source text is ignored because a syntax check is usually of no benefit. However, if you require a listing with the included files, give the directive clause **noignore include** as a switch on the schedule line.

To include listings of other sources, for example, command files, UPS has a **nosyntax** option. When you use the **nosyntax** switch, the listing should appear last on the schedule line.

Configuration

A customized message file, UPSA.MSG, configures UPS. The message file can be a shared file for all users or a private file within a specific directory. You configure UPS with a message editor called MED. Editing is allowed on the list banner and the default settings for program options **break**, **length** and **width**. Editing is also allowed on the configuration of UPS to recognize switches of more than

Figure 3

```
1)  To check for syntax errors:-

        UPS myprogram

2)  To obtain a listing:-

        UPS myprogram/list

3)  To obtain listing with page breaks
    at new procedure definitions and bodies:-

        UPS myprogram/list/break=20

4)  To change the width and length
    of the listing:-

        UPS myprogram/list/length=80/width=100

5)  To obtain a listing with included files:-

        UPS myprogram/list/noignore include
```

Figure 4

break=n	Sets the automatic break on procedures, functions and their respective 'bodies' to 'n'.
errors=n	Sets the error tolerance to 'n'. The syntax checker will exit after 'n' errors.
length=n	Sets the length of the list page to 'n'
list	Request a listing to a file or device
ignore options	Ignore specified options(s). A list of directives can be specified separated by commas.
nolist	No listing
noignore option	Re-enable options(s)
nosyntax	Turn off syntax checking. This obtains a listing with no structural information.
width=n	Sets the width of the list page

one implementation of Pascal–2, for example, native and cross products on the same host.

Changes to the list banner may be useful when software must be documented for third party transfer, or when project identity needs to be evident on each listing. MED currently functions only on terminals that support ANSI escape sequences, such as the VT100 family.

You can check the configuration of UPS with a reporter called UCR, which lists the current configuration on the users terminal. Figure 5 illustrates a typical configuration listing.

Figure 5

```
UPS Configuration RSX V1.1FT Site f1312-0
Unit-C Ltd., Dominion Way West, Worthing,
  West Sussex, England

Support From:  Unit-C
Licence Date:  10-Jan-84
Expiry  Date:  10-Jan-84
Recognises Host   Switches For:
  RSX, RT11, RSTS, UNIX
Recognises Target Switches For:
  DEC, 68000
Defaults:-
  Lines Per Page  = 60
  Width of  Page  = 79
  Error Tolerance = 100
  Page Break Trip = 0
```

Product Status

UPS is implemented on RT11 and RSX (see Information Exchange). Work is in progress on a UNIX syntax checker and plans are being made for a VERSAdos version.

Pascal has a standard II

Last summer, the International Standards Organization approved the draft Pascal standard as the new official standard. But we heard that some of the "yes" votes were conditional, and it wasn't clear whether we had an official standard or not. We have confirmed (again) that yes, there is a Pascal standard.

The language is identified as International Standards Organization ISO 7185 and was published in late 1983. The ISO standard is identical to the British standard, and orders should reference the British number, BS 6192. The standard is available from ANSI for $64.00, prepaid. Request document BS 6192 and write: ANSI, International Department, 1430 Broadway, New York, NY 10018.

The ISO standard allows two levels of conformance, Level 1 (including conformant array parameters) and Level 0 (not including conformant array parameters). The American ANSI-IEEE standard is identical to Level 0 of the ISO standard.

Pascal–2 for PDP-11/UNIX conforms to Level 0. All other Pascal–2 native compilers, including 68000/UNIX, conform to Level 1.

NAMRTS utility added for RSTS

Oregon Software still includes the NAMRTS utility with the Pascal–1 and Pascal–2 compilers for RSTS. With this utility, users who are running Pascal–1 and Pascal–2 on their systems at the same time can easily identify which run-time system is associated with which Pascal program. They can then make new associations where necessary.

Because of changes DEC has made in the RSTS operating system, this utility does not work under RSTS version 8.0 or later.

For RSX Systems

Accessing I/O devices from Pascal

by Steve Poulsen

PDP-11 computers control I/O devices by device control registers, located in a special area of memory called the I/O Page. The I/O Page is in the highest 8 KB which can be addressed on the particular PDP-11. The I/O Page on a PDP-11/45 exists in addresses 760000 through 777777.

The RSX operating system usually maintains complete control of all I/O devices. The RSX executive protects the I/O Page from users' programs by means of the memory management hardware.

Each device is controlled by "device control registers", a set of one or more words located in the I/O Page. Program can access these device control registers much the same as any other words in memory, except that some device control registers are read-only or write-only. Command are sent to a device by storing a value in a device control register, and the device returns status information and data in the registers. Disks and other high-speed device return data directly into memory buffers.

Device drivers can access the I/O Page but writing device drivers is a complex task and they generally cannot be written in Pascal.

Privileged tasks on RSX also can access the I/O page. When a privileged task is loaded, the RSX executive maps the upper 8 KB of the task's virtual address space to the I/O Page. These privileged programs can be written in Pascal, but this is not recommended because privileged tasks also map over the executive. This limits the size of the task, and programs with errors can easily crash the system. Privileged programs have other side effects which you may not desire.

The best way to access the I/O Page from a Pascal program is to use a "device common." A device common is similar to a shared common area. A shared common area provides a map from a task's virtual addresses to addresses in a named partition in physical memory, while a device common provides a map from a task's virtual addresses to addresses in the I/O page. The device common can provide access to the entire I/O Page, or it can limit access to a range of as little 64 bytes. This limited access helps prevent programs with errors from causing harm to the system.

WARNING

> Extreme care should be exercised when accessing the I/O Page. If the device being accessed is "known" to the RSX system, interrupts for that device should be temporarily disabled to prevent the device driver from receiving unexpected interrupts.

The following example, executed on a PDP-11/45, demonstrates how to access the KW11-P programmable clock from a Pascal program. The three clock control registers are located at addresses 772540, 772542, and 772544.[1] The program creates a device common which provides access to addresses 772500 to 772577 in the I/O Page. This range, which contains the clock device registers, is the smallest range which can be mapped with a device common.

As in the case of a shared common area, a dummy file creates the Task Builder data structures required to access the device common. You are mapping 100 bytes, so a simple MACRO file that allocates 100 bytes can be used.

```
.title PCLOCK Programmable clock
              device common
.BLKB 100
.end
```

You cannot use this file to initialize the I/O device itself. Its purpose is to create a Task Builder symbol table file. The macro assembler produces the object module PCLOCK.OBJ from the file PCLOCK.MAC.

```
>MAC PCLOCK=PCLOCK
```

The Task Builder is then used to produce the symbol table file.

```
>TKB
TKB>PCLOCK/-HD,PCLOCK,PCLOCK=PCLOCK
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=PCLOCK:160000:100
TKB>//
```

The /-HD option specifies that no header should be created for the task. The STACK=0 option prevents space from being allocated for a stack. The PAR=PCLOCK:160000:100 option specifies use of the device common area called PCLOCK. 160000 is the virtual address at which the device common appears in each task which maps to the device common. As with a shared common area, the base address must be a multiple of 8K bytes. 160000 was used as the virtual address because it is the highest 8K bytes available in a 64 KB virtual address space. High virtual memory should be used because Pascal tasks extend from low virtual memory to higher virtual memory as more heap space is required. The size of the device common area is the minimum size, 100 bytes.

The creation of a device common is similar to the creation of a shared common area. This command:

```
>SET /MAIN=PCLOCK:7725:1:DEV
```

creates a device common called PCLOCK in the address range of 772500 to 772577 in the I/O Page. Note that addresses and lengths are expressed as multiples of 100 bytes. The PAR command allows you to see the device common:

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 515700 MAIN SYS
       041474 242100 020000 SUB  (...MCR)
PCLOCK 041554 772500 000100 MAIN DEV
```

[1] A complete description of the KW11-P clock can be found in the "PDP-11 Peripherals Handbook."

The device common is made available when PCLOCK.EXE is installed with the INS command:

```
>INS PCLOCK
```

The INS command does not copy PCLOCK.EXE into the device common area. It only makes the device common known to the system and makes it possible for the Task Builder to link tasks to the common area.

When a Pascal task maps to the device common, virtual addresses 160000 to 160077 in the Pascal task are mapped to addresses 772500 to 772577 in the I/O Page. As a result, the clock registers, which start at 772540 in the I/O Page, are located at virtual address 160040 in the Pascal task. In other words, the clock registers at 772540 are offset by 40 bytes from the base of the device common at 772500. Therefore, in the Pascal task, the clock registers are located 40 bytes past the virtual base of the device common (160000) in the Pascal task.

The following program initializes and starts the programmable clock at a 100 kHz rate. It then samples the clock twice to determine how much time was required to compute a natural logarithm.

```
Program Clock;

{  Example showing how to access I/O devices
   from Pascal }

  const
    clock_address = 160040B; { Clock virtual
        address }
  type
    kw11p_csr =
      PACKED RECORD
      { KW11-P Control/Status Register }
        run: boolean; { Start counting }
        rate: (khz_100, line_freq, khz_10,
              external_freq);
        mode: (single_interrupt, repeat_interrupt);
        direction: (down, up);
        fix: boolean;
           { For maintenance operation }
        interrupt_enable: boolean;
        done: boolean;
           { Done counting }
        unused: 0..127;
           { Unused bits in CSR }
        error: boolean;
           { Error detected }
      end;

    clock_registers =
      RECORD
        csr: kw11p_csr;
          { Control/Status register }
        count_set_buffer: integer;
        counter: integer;
      end;

    clock_pointer = ^clock_registers;

  var
    clock: clock_pointer;
     { Pointer to clock control registers }
    start, stop: integer;
     { Start and stop counter values }
    r: real;

  begin
    clock := loophole(clock_pointer, clock_address);
    WITH clock^.csr DO
      BEGIN { Initialize the clock }
      rate := khz_100;
        { 100 kHz clock rate }
      mode := single_interrupt;
      direction := up;
      fix := false;
      interrupt_enable := false;
      end;
    clock^.csr.run := true;
      { Start the clock }
    start := clock^.counter;
    r := ln(1.23456789);
    stop := clock^.counter;

    writeln('Time: ', (stop - start) *
        10: 1, ' micro-seconds');
  end.
```

The record KW11P_CSR defines the clock control and status register which maps the fields in the control register. (See the "PDP-11 Peripherals Handbook" for a definition of the control register.)

The type CLOCK_REGISTERS defines the three clock device registers. The type CLOCK_POINTER is defined as a pointer to CLOCK_REGISTERS, and the variable CLOCK is defined as a CLOCK_POINTER. The variable CLOCK does not allocate the clock registers; it is simply a one-word address.

The connection between the Pascal program and the I/O Page is established when the LOOPHOLE function in the first statement in the program assigns the address of the clock registers (the constant CLOCK_ADDRESS) to the pointer CLOCK. The pointer CLOCK is assigned a virtual address

of 160040. When the program is linked with the device common, virtual address 160040 in the task is mapped to physical address 772540 in the I/O Page. When the program is running, references through the pointer CLOCK access fields in the clock's device registers.

Compile the program with the Pascal-2 compiler. (Pascal-1 users must modify the program to change the packed record and use a variant record to assign a value to a pointer.) When the clock task is built, the device common is treated the same as any other shared common area.

```
>PAS CLOCK
>TKB
TKB>CLOCK/FP/CP,CLOCK=CLOCK,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>RESCOM=PCLOCK/RW
TKB>//
```

The RESCOM option (above above) causes the Task Builder to read PCLOCK.STB to obtain information needed to map the PCLOCK device common . If PCLOCK.STB and PCLOCK.EXE are copied into LB:[1,1], then instead of the RESCOM option, the COMMON=PCLOCK:RW option could be used.

When the program is run, you see the time required to compute a natural logarithm.

```
>RUN CLOCK
Time: 260 micro-seconds
>RUN CLOCK
Time: 250 micro-seconds
>RUN CLOCK
Time: 250 micro-seconds
```

With the clock counting at 100 kHz, each "tick" represents 10 micro-seconds. This means that the total time is computed by counting the number of ticks and multiplying by 10.

Although this example may seem simple, it has a side effect on the RSX system on which it runs. Pascal programs cannot easily handle device interrupts because the interrupts are processed in kernel mode by highly specialized code. In this example the clock interrupt enable bit is turned off so that you have stopped time on the system:

```
>TIME
22:47:37 09-FEB-84
>TIME
22:47:37 09-FEB-84
>TIME
22:47:37 09-FEB-84
```

RSX uses the clock to keep track of the time of the day. When the clock interrupt is disabled, RSX loses track of the clock. To fix the problem, use the OPEN command to alter the clock control and status register. The following example shows how to re-enable clock interrupts at the proper rate in the proper mode:

```
>OPE 772540
772540 /000220 115<ESC>
```

The value 115 was computed from the information about the clock in the "PDP-11 Peripherals Handbook." It is easier to examine the clock status register to determine the correct control register settings before running the Clock program.

```
>TIME
22:47:49 09-FEB-84
>TIME
22:47:53 09-FEB-84
>TIME
22:48:00 09-FEB-84
```

The clock now seems to be in order, but it may be a good idea to re-boot the system to be sure nothing else has been damaged.

As this example shows, extreme care must be taken to temporarily disable interrupts to the device being accessed, if the device is "known" to the RSX system.

Do not be tempted to try to perform special terminal I/O by directly addressing the terminal's device control registers. With the wide range of control provided by the SET command and the subfunction bits available with the QIO system directive, direct access to terminals should seldom be necessary. The best policy is to only access devices which are not part of the RSX configuration.

I/O switches enhance your terminal's performance, without penalties

by Steve Poulsen

The RSX terminal driver provides several automatic features which may not suit special I/O applications. Using combinations of I/O control switches added in Pascal–2 Version 2.1A, you can control these automatic features without the performance penalties of previous solutions.

During normal operation, the terminal driver itself supplies the line formatting characters: it first prints a line-feed character to skip to a new output line, writes the data in the line to the terminal, then adds a carriage-return character. The terminal driver also automatically inserts carriage-return/line-feed characters when an output line "wraps" around the right edge of the screen or paper. It provides editing functions which delete input characters or entire lines and expand tab characters to spaces.

For some applications, you may prefer to use direct cursor addressing to place output at specific locations on a terminal's display screen. The buffering performed by Pascal and the terminal driver may prevent the output from being printed when you want it to be. Since the terminal driver inserts a line-feed at the start of the line and a carriage-return at the end of the line, it may not be possible to write data to the last line on the screen, for example. The line-feed will cause the entire screen to scroll up.

You may have tried any of three methods to alter terminal driver activity. Each has its drawbacks. The RSX `SET` command controls many of the terminal driver features. However, these changes are permanent until a subsequent `SET` command is issued. As explained in the "Pascal–2 V2.1/RSX User Manual," you can specify the `/FTN` switch and then use the FORTRAN carriage control conventions. But the `/FTN` switch does not cause data to be written to the terminal immediately. The characters are still buffered until a `writeln` is executed, causing annoying delays in terminal print out. A third alternative, also in the user manual, is controlling the buffering of input and output with the `/BUFF:nnnn` switch. System performance suffers greatly with the `/BUFF` feature because the program must communicate with the terminal driver for each character read or written. In addition, the input editing features are disabled because the terminal driver is not buffering the input.

The following combinations of Pascal–2 I/O switches often obtain all of the control required without the performance penalties of previous three methods.

/RAL/BUFF:1 and /WAL/BUFF:1 Switches

The `/RAL` (read all bits) switch prevents the terminal driver from interpreting any input characters, such as the delete character (which normally deletes the previous input character) and control-U (which normally deletes the entire line). All characters typed are passed to the program. The `/WAL` switch (write all bits) disables the automatic line wrap feature. The combination of `/RAL/BUFF:1` on input and `/WAL/BUFF:1` on output gives you complete control of all terminal I/O. The terminal driver passes all characters typed by the user to the program and each character in a write statement is sent immediately to the terminal with no special interpretation.

/NOCR Switch

With the `/NOCR` switch, you can disable the automatic insertion of carriage control characters by the terminal. When you use this switch, a `writeln` statement writes the line without any additional carriage control characters. You must insert carriage control characters with the `CHR()` function. This switch is most often used with the `/WAL` switch to prevent the automatic line wrap feature. Use the combination `/NOCR/WAL` when performing direct cursor addressing on a video terminal.

/NOECHO and /RNE Switches

The terminal driver normally echoes input typed by the user. You can disable this feature with the `/NOECHO` or `/RNE` (read with no echo) switches. These switches must be applied to the input file as shown in program RNE:

```
Program RNE;
    var
      pwd: PACKED ARRAY [1..10] OF char;
    procedure Getpassword;
      var
        inp: text;
      begin
        write('Password: ');
        break(output);
        reset(inp, 'TI:/noecho');
        readln(inp, pwd);
        close(inp);
      end;
    begin
      Getpassword;
    end.
```

The BREAK() procedure forces the output of the partial prompt line. You must use it when a partial line is to be printed and either the input file is not the standard input file or the output file is not the standard output file. Normally this procedure is not required because an implicit BREAK(OUTPUT) is performed when input is read from the standard input file.

/RST Switch

Used with an input file, the /RST switch terminates the input line when any non-printing character (any control character) other than a space is typed. You can identify this termination character by examining the internal file structures used by the Pascal support library. The program RST shows how to use the /RST switch. The include file LIBDEF.PAS is used to define the internal support library data structures.

```
Program RST;

    %include 'libdef';

    var
      line: PACKED ARRAY [1..80] OF char;

    function Terminator(VAR f: text): char;

      var
        x: user_file_variable;

      begin { Terminator }
        x := loophole(user_file_variable, f);
        terminator := chr(x^.iosb DIV 256);
      end; { Terminator }


    begin { Main }
      reset(input, 'TI:/RST');
      write('Type a line: ');
      readln(line);
      writeln('ORD(terminator) = ',
         ord(terminator(input)): 1);
    end. { Main }
```

You can use the terminator function defined above with any input file to determine which character terminated the input line. The terminator is always present in the I/O Status Block field, even if the /RST switch is not used.

/CURSOR Switch

The /CURSOR switch enables terminal-independent cursor control. When you open a terminal as a file with the /CURSOR switch, the first two characters on each line are used to specify a column and line number at which the text is to be printed. The terminal driver generates the necessary cursor control sequences to position the cursor. This means that a Pascal program can perform cursor positioning and graphics on a variety of terminals without having to be recoded for each different terminal. (Terminal-independent cursor control is a SYSGEN option, so it may not be present on all RSX systems.)

The first character on each line is the column number (horizontal position) with column 1 being the first column at the left. The second character is the line number, with line 1 being at the top of the screen. If 128 is added to the line number, the screen will be erased before the line is displayed. The program Spiral shows how the /CURSOR switch can be used.

```
Program Spiral;

{ Demonstration of terminal independent
 cursor control }

  const
    increment = 0.1;
    decay = 0.99;

  var
    angle, radius: real;
    x, y: integer;
    f: text;


  begin
    rewrite(f, 'TI:/CURSOR');
    writeln(f, chr(1), chr(129));
    radius := 12.0;
    angle := 0.0;
    WHILE radius > 1.5 DO
      begin
      x := round(2.0 * radius *
           sin(angle)) + 24;
      y := round(radius * cos(angle)) + 12;
      writeln(f, chr(x), chr(y), '*');
      angle := angle + increment;
      radius := radius * decay;
      end;
  end.
```

/RCU, /WBT, and /PR:0 Switches

In the previous example, the /RCU switch restores the cursor position. When you use this switch, the terminal driver saves the current coordinates of the cursor, prints the line,

and then restores the cursor to its original position. This switch is most often used with the /CURSOR switch to update a field on a CRT without disturbing input or output in progress elsewhere on the screen.

The program Timer shows how the /RCU and several other switches can be used. The program detaches from the terminal and then, every five seconds, updates the time of day in the upper right corner of the screen. The /CURSOR switch specifies the location of the time, independent of the terminal being used. The /RCU switch causes the terminal driver to restore the original cursor position after updating the time. (The /CCO switch disables control-O mode before the time is updated and is not essential to this example.)

```
Program Timer;

{  Print time display on terminal }
  var
    mag, unit, h, m, s: integer;
    t: real;

  procedure Wait(VAR magnitude, unit: integer);
    NONPASCAL; { Defined in system library }

  procedure Detach;
    EXTERNAL; { Defined in Pascal library }

  begin
    detach;
    mag := 5;
    unit := 2; { seconds }
    rewrite(output, 'ti:/cco/rcu/cursor/wbt');
    writeln(chr(1), chr(129));
    repeat
      write(chr(66), chr(1));
      t := time;
      h := trunc(t);
      t := (t - h) * 60.0;
      m := trunc(t);
      t := (t - m) * 60.0;
      s := trunc(t);
      write('  ', h: 2, ':');
      IF m < 10 THEN write('0');
      write(m: 1, ':');
      IF s < 10 THEN write('0');
      writeln(s: 1, ' ');
      wait(mag, unit);
    UNTIL false;
  end.
```

The /WBT switch selects "break-through write" mode. This displays output regardless of the current terminal state. If this switch were not used, the time would not be displayed when a program attached to the terminal. The /WBT switch is not necessary, and you may not want to use it because of possible side effects with screen-oriented editors. The /WBT switch requires that the task be privileged. The /PR:0 switch in the Task Build line makes the Pascal program privileged (without overmapping the monitor, which is unnecessary in this case).

To build and run program Timer, use the following commands:

```
>PAS TIMER/NOWALKBACK
>TKB TIMER/FP/CP/PR:0=TIMER,LB:[1,1]PASLIB/LB
>INS TIMER/TASK=...TMR
>TMR
```

At this point, the current time should appear in the upper right hand portion of your screen. If you do not wish to use break-through mode, you can remove the /WBT switch and the /PR:0 switch from the Task Builder command line. Every five seconds, the time is updated. To stop the time display, use: >ABO TMR.

Oregon Software expands...

In the last few months, we have added several new people. We'd like you to know who they are and what they do.

Joel Clark joined Oregon Software in May as a programmer. He got started with computers using a DEC Rainbow for accounting, bidding, and word processing at his own construction company, then studied at Computer Careers Institute and Portland Community College. Joel says he is enjoying learning the UNIX system and the 68000 series. His other interests include hiking and sailing.

Sales representative **Penny Green** brings a varied background to her new position: she wrote the sports column for a local newspaper and has sold real estate and advertising. Penny has a bachelor's degree in mathematics from Portland State University and is now in the M.B.A. program. She enjoys camping and reads history and theology.

Lisa Payne is on the front line as our new receptionist. Lisa previously headed the publicity department of a manufacturing firm and has a strong background in graphics. She started in February this year and says she appreciates the congenial atmosphere of her new position.

Steve Hampson, Senior Software Engineer, earned his M.S. in computer science at Ohio State University. Before

coming to Oregon Software, he spent five years with the language development group of an Ohio firm, where he maintained the extended FORTRAN compiler. Steve says he enjoys the Oregon mountains and trees and plans to explore his new state through photography, skiing, and hiking.

As Manager of Customer Service, **Claire Lematta** helps customers with all phases of the ordering process and handles software support sales. She maintains the field test and software licensing agreements and can answer non-technical questions about them. Claire enjoys our varied customer base and is especially qualified to help overseas customers. She has lived all over the world, speaks Spanish, and has a degree in international studies. Claire is, however, a native Oregonian who enjoys cross-country skiing and backpacking as well as participating in the Oregon chapter of the World Affairs Council.

Mary Erichsen, OEM Manager, has been a senior sales representative at Wang and was a large account manager at Victor Technologies. At Oregon Software, she provides sales tools, workshops, and competitive information for existing and potential OEM customers. She recently negotiated agreements between Oregon Software and some top corporations in the industry. Mary travels throughout the United States to call on customers and to attend trade shows. Recently she broke her ankle while running in Boston, but she's still running. Be on the lookout for Mary at trade shows.

Tom Hanrahan joined Oregon Software in February as a technical writer. After obtaining his M.S. in engineering at U.C.L.A., he worked as a free-lance technical writer for three years. Now he is producing new manuals and updating old ones. Tom is another runner; he competes on weekends, if he's not cross-country skiing.

Hanh Hoang, our new data entry operator, puts ordering information from customers into our files. Hanh has her certificate of data entry operation from Portland Community College. When she's not working, Hanh enjoys dancing.

Gerry O'Scannlain is responsible for Oregon Software's advertising campaigns. She recently produced our handsome SourceTools package. Her next major campaign will be for Pascal–2. She brings extensive experience in marketing, graphics, and fund raising to her new job. Customers will be seeing Gerry's work at trade show presentations and as eye-catching, high quality ads in trade journals.

Louise Waitt recently joined the writing staff as an apprentice technical writer. In addition to her B.A. from Oregon State University, she earned a certificate in computer science at Denver University and worked in Colorado as a programmer. When her training is finished, Louise will be producing and maintaining OSI's technical manuals.

Dirk Eide began as a software engineer this February, after four years of experience in software and systems engineering as well as graduate coursework in math. Dirk is maintaining the RT-11, RSX, PDP-11, and UNIX systems for Oregon Software. His avocation is riding race horses at Portland Meadows Race Track.

Mark Paulin also joined our programming staff this February. He is modifying Pascal–2 compilers to run under UNIX on the 68000 series. Mark comes to Oregon Software after receiving his B.A. in math from Reed College and spending three years at Tektronix, where he designed and maintained software products.

Continued on Page 23

Pascal NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

David Spencer, editor
Ann Littlewood, Thomas E. Hanrahan, writers
Jennifer Mulder, production assistant

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 2340 SW Canyon Rd., Portland, OR 97201; (503) 226-7760. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of the editor. We pay $100 per newsletter page for any article we print.

Printed in USA

The Log: Pascal-1 and Pascal-2

Oregon Software's previous Pascal Newsletter described significant changes in Pascal-2 Version 2.1B. This log details bugs fixed in Version 2.1C, now available.

Pascal-2

Version 2.1C corrects these problems for PDP-11s with DEC operating systems.

ORIGINed variables misplaced

Addresses allocated to variables through the **origin** declaration were sometimes improperly assigned and led to incorrect code generation.

Set operation errors

The inclusion operator, **in**, generated bad instructions when certain elements were not recognized as belonging to a set.

Long comparison breakdowns

Particularly long, involved comparisons sometimes created an **undeleted temps** error and resulted in compiler shutdown.

'Exp' and 'Sqrt' problem

Under double precision, an inadequate range-checking routine for arguments sometimes caused **exp()** to return an incorrect value. Double precision **sqrt()** produced the wrong result when its argument was 0.

'Rad50' routines didn't work

Rad50 routines omitted the letter 'A' from the first and fourth positions of file names they were required to print. For example, **Rad50** would print the file name "HDRA.PAS" as "HDR.PAS."

String package miscues

Several logical and typographical errors in the string package have been corrected. In addition, two procedure names, **AssChar** and **DelString**, have been shortened to allow the entire string package to be placed in an external library.

Switching errors in /DEBUG option

Attempts to use the **/debug** option on external modules produced a spurious, **conflicting switches** error. With version 2.1C, you can debug external modules.

Changes to RSX

Version 2.1C for RSX corrected these problems:

The error in 'noioerror'

The **noioerror** procedure generated an incorrect valu when reporting the status of errors involving the **put(** operation for random access files.

/LUN:n switch sensitivity

The **/lun**:*n* switch, which is used to give a file a logica unit number other than the default assigned by Pascal-2 has been adjusted so that it is no longer sensitive to th order in which switches are applied.

'Forini' incompatibility

The FORTRAN initialization routine, **forini**, failed t work properly on VAX in RSX compatibility mode.

Oversized buffer files incompatible with SYSLIB

The support library no longer allocates large buffers whe files containing oversized records (greater than 512 bytes are opened. Previously, users were forced to use ANSLI instead of SYSLIB. Pascal now allocates 512 byte buffer for all disk files unless the **/buff**:*nnnn* switch is usec (Users taking advantage of the **/buff**:*nnnn* switch mus remember, however, that creating record sizes greater tha 512 bytes again forces the use of ANSLIB.)

Changes to RT-11

Version 2.1C for RT-11 corrects these problems:

SIM errors

The simulation mode switch (SIM) for floating point aritl metic incorrectly handled **text** files and made the debu ger inoperable.

'Rename' lacked a default

Rename now uses any field of the old file name that tl new file does not specify as a default. Other, intermitte errors in **Rename** were also corrected.

Stack checking errors

Stack checking errors associated with completion routin written in Pascal have been eliminated.

Changes to UNIX

Version 2.1D corrects these problems for UNIX/68000 systems.

Code generation corrected

The compiler no longer generates incorrect code for operations with sets or for accessing structured constants fields of packed records.

Internals error corrected

Certain complex expressions no longer cause the compiler to generate the internals error.

Version 2.0N corrects these problems for UNIX/PDP-11systems.

PDP-11/23 problems fixed

The compiler now runs on the PDP-11/23; however, some features of the Debugger are still not available.

Source Tools

Version 1.0C corrects these problems.

'Newsrc' no longer prompts unnecessarily

`Newsrc` does not prompt for information that you've already supplied on command lines such as:

```
newsrc/input=test.pas/release=1/key=(descr="base
line") mymodule
```

In previous releases, `newsrc` prompted for the module name `mymodule`.

'Make' functions corrected

The command

```
make test/debug=foo
```

now generates FOO.DBG instead of building the file MAKEFILE.DBG.

The command

```
make test/hier
```

no longer dies with a run-time error.

Changes to RSTS/E

Version 2.1C for RSTS corrects these problems:

Real numbers misread

When consecutive real numbers were to be read, the first character in the series was missed.

Run-time errors not distinguished

Run-time error checking routines now provide negative error numbers for operating system errors and positive ones for error messages from the Pascal support libarary.

Sayerr error

Sayerr now does not print, `RSTS error...` before the body of the error statement.

Known bugs

Our last newsletter promised customers a statement of company policy on known bugs. Future sets of release notes will contain a list of unfixed bugs and workarounds, if known. We will also print this information in the newsletter, starting with the next issue. We hope these procedures will spare customers both the annoyance of discovering a bug and the unnecessary labor of reporting an already known bug.

Errors, additions to manuals

The RSTS Pascal 2.1 manual, page 2-59, omits a command line essential for calling FORTRAN from Pascal. After the command line:

```
LINK FTEST=FTEST,INNAM,P:PASCAL,$FORLIB
```

as listed in the manual, add:

`PRTS FTEST n` (where n is some memory size 2 to 28).

This command allocates enough memory for the program to run and associates the Pascal runtime system with the .SAV file. Otherwise the program tries to run with the RT-11 runtime system, the "default" for FORTRAN and for programs that are built on RSTS using the `LINK` command. Without the second command line, the program will get an "M-Trap to 4" error immediately and die.

Case Study

Programming a process-control system

by Kurt W. Papke

Mr. Kurt W. Papke was formerly the Engineering Manager at Spectronix, a small systems house that writes software for real-time process control systems for handling bulk materials, primarily for the grain industry. In this article, he describes the trials and tribulations of programming one of Spectronix's early projects, then relates the scope of an even larger current project.

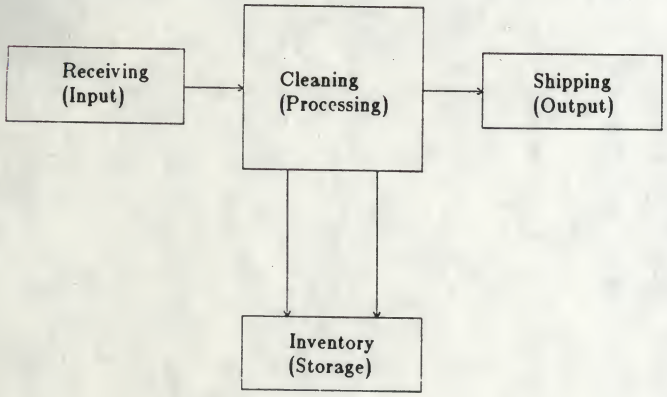
Real-time industrial process control systems, with their huge databases, mulitple concurrent processes, complex control algorithms and immense quantities of I/O points, have an innate tendancy to tax hardware, software, and programmers to their limits.

The Ups and Downs of a Grain Elevator

The modern grain elevator is a complex system that handles grain much like computers handle data. In computer terms, an elevator is a large I/O buffer. Terminal elevators "buffer" domestic grain for export overseas to our friends in Russia, China, etc. Commodities are input from rail cars, barges, or trucks, then held in storage bins, and finally output to ships.

The mechanism for implementing this big I/O buffer is a mechanical engineer's nightmare. A typical grain elevator consists of conveyors, valves, modulated gates, elevating legs, and electronic scales, not to mention several hundred bins. The investment of several hundred million dollars of capital investment represented by this equipment easily justifies the expense of a computer control system.

Figure 1. Elevator as I/O Buffer



Conceptually, a grain elevator looks like a large directed graph or "digraph." The edges of the digraph are made up of the conveyors and elevating machinery, and its nodes are the holding bins and switching equipment. It is the job of the computer system to control, monitor and report on the movement of the grain through this digraph. Consider the flow diagram of a typical terminal elevator shown in Figure 2.

The control system directs the grain through the elevator by controlling solenoid or hydraulic switching mechanisms at each "fork in the road." It must be careful to avoid collisions of grains from different sources. Extensive exception handling has to be built into the logic to cope with situations like bins filling and equipment failure.

Typical Process-Control Architecture

A typical control system for this industry uses a classic three-level hierarchy (shown in Figure 3).

The top level preforms supervisory functions for the entire plant, making strategic decisions and serving a data base function. Typical machines at this level are of the VAX class, due to the data base orientation of their function.

The second level of the hierarchy controls a specific area of the physical plant such as a rail-receiving stream. The processor at this level makes local tactical decisions, usually based upon a massive amount of input. Typical machines at this level are PDP/11-44s, chosen for their high I/O handling capability and memory capacity.

At the peon level lives the Programmable Logic Controller, a strange beast manufactured by the likes of Modicon, Allen-Bradely and Texas Instuments. They are programmed in the relay-ladder logic and are particularly well-suited to interfacing to the world of motor starters, rotary encoders, and limit switches.

Why Pascal?

The vast majority of process-control applications are coded in a combination of assembler and FORTRAN. Assembler is used for "efficiency" (an illusory metric, of concern mostly to the person who purchased too little memory for the CPU). FORTRAN is used because process-control systems are designed by engineers, who only learned FORTRAN in college, and who think that the only construct required in a programming language is an arithmetic IF.

Figure 2. Example of an Elevator Digraph

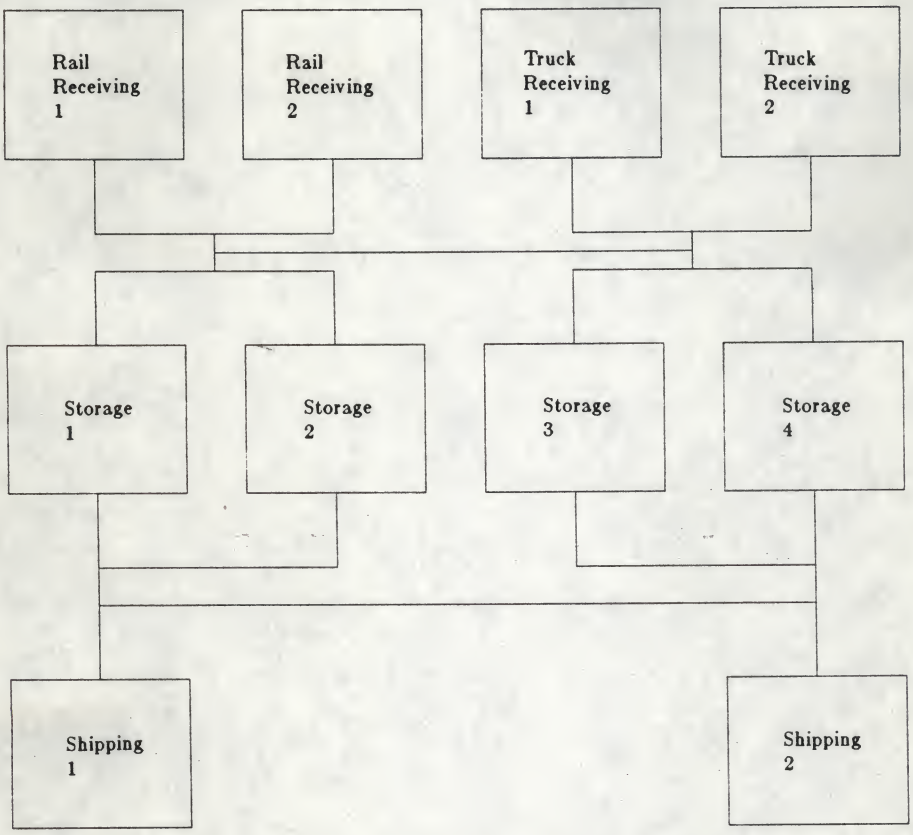
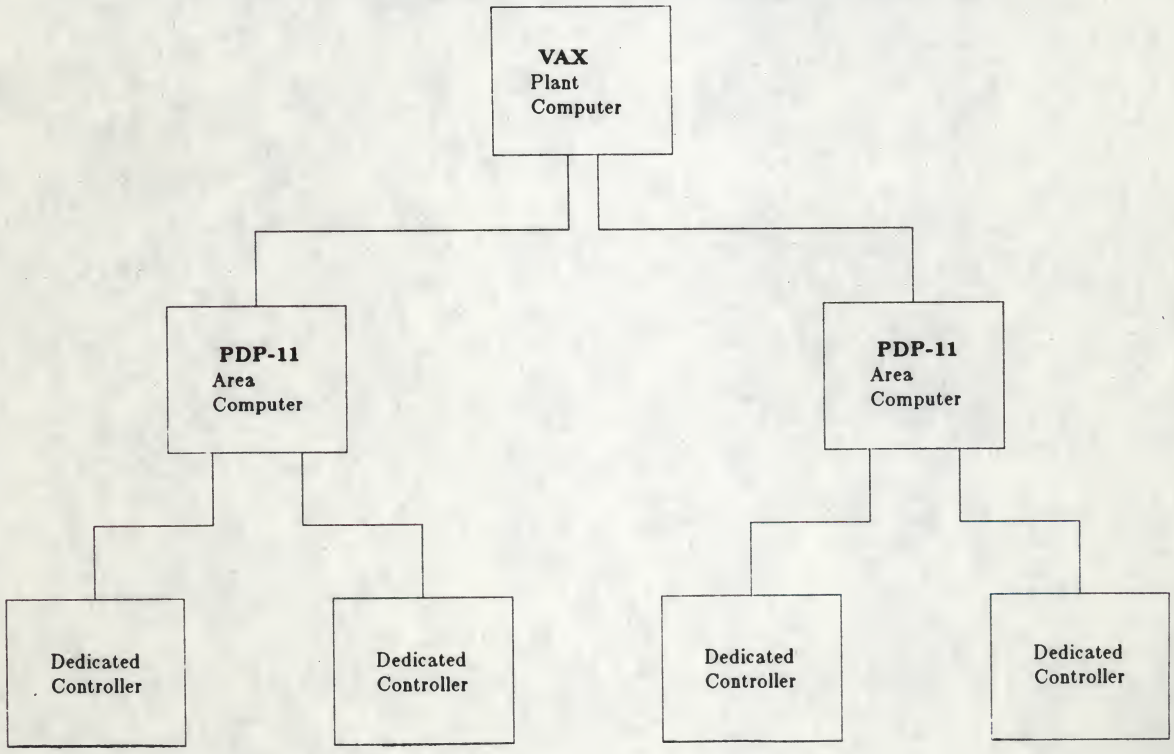


Figure 3. Classical Control System Design



We chose to use Pascal because of the extremely complex nature of both the algorithms and data structures we must deal with, such as digraphs, shortest path, linear programming, etc. A typical shared common region, which is the memory-resident data base that drives the control system, is made up of arrays of records of arrays of records of....

Such complex structures are awkward to handle in assembler or FORTRAN. Another factor is the Mongolian Horde of Pascal programmers graduating from the universities; it is becoming far easier to find Pascal programmers than practitioners of any other language.

So far the decision seems to have been a wise one. The language has some major deficiencies, but considering the alternatives, Pascal appears to be the least evil. Other languages considered as a replacement were Modula-2 and Ada.

An Early Effort

The first large system we designed was for a Gulf Port export elevator. The system was intended to replace the traditional control panel consisting of lights and switches with color graphic displays and a supervisory minicomputer.

The computers were a bit undersized: MC6809 processors for the tactical level and a single PDP-11/44 for the supervisory and data base machine. The 11/44 was equipped with 1 megabyte of RAM, dual RK07 drives and 32 serial ports (gasp!). The system software consisted of RSX-11M V4.0, RMS-11K, Datatrieve, FMS-11 and Oregon Software's Pascal-1. We considered converting to Pascal-2, but abandoned the idea because of the enormous effort such conversion required.

The applications software consisted of roughly 75 Pascal programs in 40 directories making up 50,000 lines of code (comments included). The design and coding effort required 7 man-years in a 16 month calendar period using 5 programmers. If you work out the numbers, this averages out to 30 debugged and documented lines of code per day per programmer, versus the supposed industry average of 10 lines. In terms of actual dollars, this software costs roughly $10 per line of debugged code.

In order to complete the project, we had to deal with a number of issues concerning the use of Pascal-1.

- Source code sharing
- Calling RSX directives
- Writing asynchronous system trap (AST) routines in Pascal-1
- Use with FMS-11
- Use with RMS-11K
- Use with multiple system resident common blocks
- Symbol table space shortages

Our problems were caused by: difficulties inherent in Pascal itself, the FORTRAN orientation of RSX-11, and the constraints imposed by Pascal-1, many of which have been eliminated in Pascal-2.

Source Code Sharing

In any large software system, there is a need to share common source code, notably utility routines and data type definitions. Unlike Pascal-2, the Pascal-1 compiler has no dynamic **%include** directive. We had a choice of two methods for working around this problem: code up a utility program to do the inclusion or use a DEC standard utility in a non-standard manner.

We decided to use SLP because it was a very reliable, DEC-supported utility. We had enough applications software to worry about without the additional concern of utility programs. The SLP input file for a program looked like:

```
PROG.PAS/-AU=
PROGRAM Prog;
@filename1.pas
@filename2.pas
BEGIN
END.
/
```

SLP processes this file to produce PROG.PAS, bringing in all files indicated by the **@** signs at up to three levels of indirection. This technique has worked out quite well and we continue to use it with other utilities that do not support dynamic inclusion, such as Oregon Software's PROSE text formatter.

Calling RSX directives

In any real-time system, programs must access the operating system directives. DEC supports only MACRO-11 and FORTRAN interfaces to the RSX-11M directives. The FORTRAN callable routines are difficult to use from Pascal because they do expect the FORTRAN run-time system to be present. They are also somewhat restrictive and do not take advantage of any of Pascal's data structuring capability.

We created an object library of Pascal-callable RSX directives, some of them designed to exploit Pascal structures. For example, we changed the "wait for logical OR of event flags" (WTLO$) to a "Wait for any in a set of event flags,"

in order to hide the use of the event flag mask. In this form the directives turned out to be more readily understood by Pascal programmers. An example call might be:

```
Const MaxEfn = 64;

Type EfnRange = 1..MaxEfn;
     EfnSet = Set of EfnRange;

Procedure WaitForSet (Efns : EfnSet); External;
```

Writing AST Routines in Pascal-1

Due to the large amount of send/receive data and intertask communication involved, we needed a method to write receive-data ASTs in Pascal. So, we wrote a short MACRO routine that took as a parameter the name of the procedure to be called for AST servicing. An "envelope" routine was specified in the SRDA$ call, which invoked the MACRO procedure, set up R5 for the heap pointer, and executed the AST exit (ASTX$) upon completion.

This worked out quite well, allowing the routine to be completely coded in Pascal. The service procedure could communicate with the main code via event flags or global variables. The code for the "Specify Receive Data AST" routine was:

```
Procedure RsxSrda (Procedure AstRtn);

    {---(MACRO Envelope Routine)---}

{$C       .IDENT "820311"     ; Creation date
$$R5$$:   .Word 0             ; Heap address
          .Mcall Astx$
AstEnt:   Mov    R0, -(Sp)    ; Save Register
          Mov    R1, -(Sp)
          Mov    R2, -(Sp)
          Mov    R3, -(Sp)
          Mov    R4, -(Sp)
          Mov    R5, -(Sp)
          Mov    $$R5$$, R5   ; Get heap pointer
          Jsr    Pc, 0(pc)+   ; Call the procedure
AstAdx    .Word 0              ; Routine Address
          Mov    (Sp)+, R5    ; Restore registers
          Mov    (Sp)+, R4
          Mov    (Sp)+, R3
          Mov    (Sp)+, R2
          Mov    (Sp)+, R1
          Mov    (Sp)+, R0
          Astx$s              ; Exit from AST
}

    {---(Actual Procedure Body)---}

Begin {RsxSrda}
    {$C
          .Mcall Srda$s
                              ; Remember heap pointer
          Mov    R5, $$R5$
                              ; Store routine address
          Mov    2(sp), AstAdx
          Srda$s AstEnt       ; Specify entry point
    }
End; {RsxSrda}
```

Use with FMS-11

Using Pascal-1 with FMS-11 is one of the most frustrating experiences imaginable. The problem has two facets: Pascal-1 has no variable length string facility, and FMS requires ASCIZ strings (terminated with a null character). The frustration level is such that, after writing every FMS task, we'd swear up and down never to use Pascal again!

There is no clean solution to this problem. If you use the Pascal-1 string package, you still must hassle with string constants, which necessarily occur everywhere in an FMS task, for field names, form names, etc.

The best we could do in Pascal-1 was to define the FMS procedure parameters to be single characters passed by address (**var**), and then fool the compiler's type checking by passing only the first character of the string. Unfortunately, the Pascal-2 compiler negates this "work-around" because string constants are **packed** arrays and an element of a **packed** array cannot be passed as a **var** parameter. Foiled again!!

The introduction of Version 2.1 of Pascal with its conformant array parameters means a solution is in sight. We are in the process now of writing a package that uses conformant arrays to pass the string parameters to FMS-11.

Use with RMS-11K

The use of Pascal-1 with RMS-11K is by now a relatively well understood procedure (see the "Information Exchange" in Pascal Newsletter Number 4). The remaining difficulties consist primarily of keeping the Pascal record definitions in synch with the Datatrieve record definitions as the fields in the files change. We've attempted to automate this process by writing a translator for Datatrieve-to-Pascal record definitions.

Multiple System Resident Common Blocks

The use of common blocks with Pascal-1 (or Pascal-2) is a simple matter: the **origin** statement can be used or an external MACRO-11 procedure can be written to set a pointer variable to the start of the common region.

One caveat however: the use of pointer variables within a system common is dangerous—care must be taken that every program maps the commons to the same spot in their virtual address space. Programs that map a common region containing pointers to different address spaces end up with odd address traps, memory protect violations, etc. Normally, these problems only occur when multiple commons are in use, and not every program specifies the APRs to be used in the task build command file.

Symbol Table Space Shortages

Due to the relatively large size of the program (when type definitions and service routines were included), running out of compiler symbol table space was a chronic problem. We never found a solution for this, other than upgrading to Pascal-2, which does not limit symbol table size to the available heap space.

The Current Effort

In the continuing saga of Spectronix versus process control, the next chapter is a large, state-of-the-art facility. We intend to provide a high-speed, fault tolerant system with the latest in man-machine interfaces.

Figure 4 shows the modern LAN-based architecture of the new system, which contains four VAX-11/750s, four PDP-11/44s, and nine Allen-Bradley PLC-3s. The DEC machines are connected via a PCL-11 (pronounced pickle) and the 11/44s talk to the PLC-3s over the A-B Data Highway. RM81 disk drives give the system more than 3 gigabytes of disk storage. All the disks are dual-ported to allow access in the event of a machine failure. Operators communicate with the system through a voice recognition system (VOTAN) and alarms are annunciated with voice output. Process status is displayed on twelve VS-11 color graphics screens, four of which are wall-projected to ten-foot diagonal displays.

The challenge is to program all those CPUs with what estimate will be 100K lines of Pascal and to solve th problems:

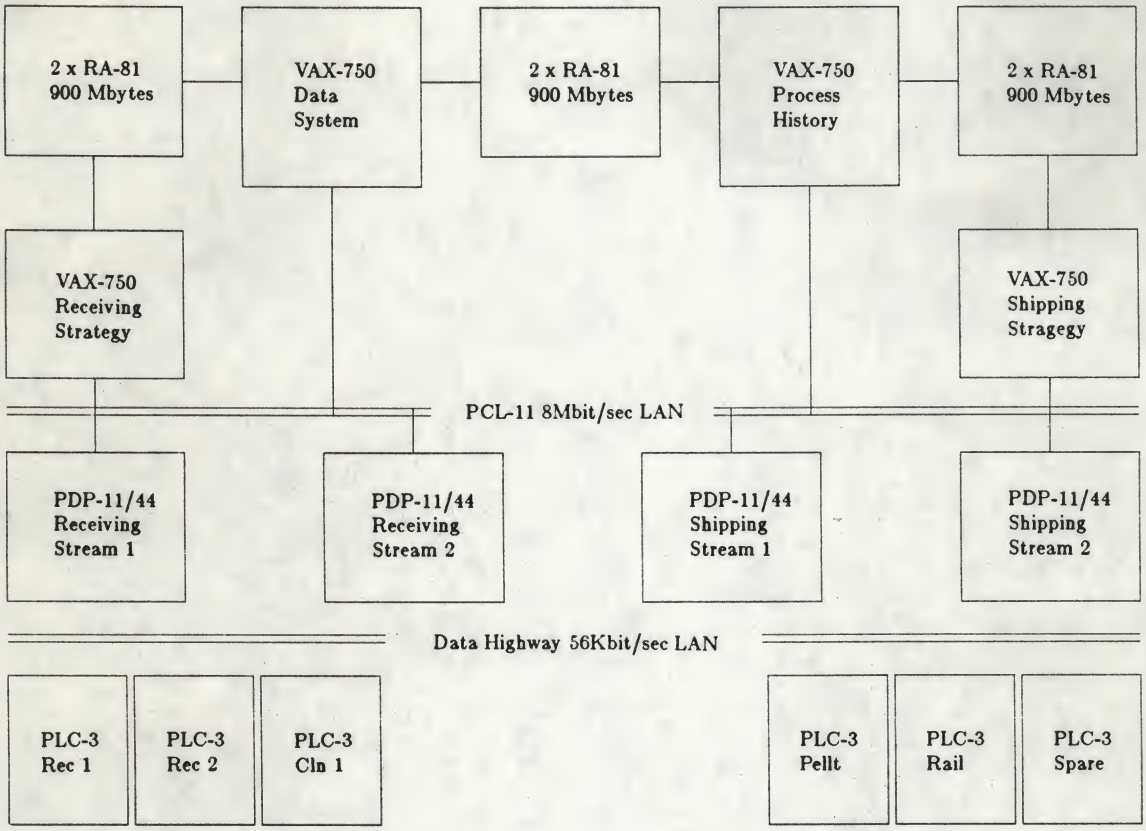
- Two different CPUs requiring two different Pascal cc pilers. It would have been nice to use an Oregon Soft compiler on the VAX but....
- Utility (FMS,RSX,RMS) libraries upgraded to run ı der Pascal-2 on the 11/44s.
- Large common blocks (>32K words) on the 11/· implemented with PLAS directives to make use se: transparent from Pascal.
- DECnet everywhere.
- Real-time graphics including process animation (are going to show the rats chewing grain in real-tin
- Voice command and response.
- Fail-safe design—if one machine goes down anot must take over screwing things up where the fai machine left off (more or less).

All this has to be done sometime in 1984. Wish us lu Lord knows we are going to need it!!

Editor's Note:

Since Mr. Papke wrote this article, a VMS version Pascal–2 is much closer to completion. Pascal–1 has limited variable-length string facility; its structure is ı generally compatible with other languages.

Figure 4. Modern LAN Based Architecture



OPUS Communiqué

Oregon Pascal Users Society
by Bruce Williams, Coordinator

The OPUS conversion from RSX-11M to RSX-11M/+ will be finished soon, followed shortly by the official opening of the OPUS library. Get ready to submit your routines!

Some members have requested the DCL/MCR Command processor routines. This will be done as soon as the library is up and running.

The membership list is in the mail. Space does not allow printing it in this newsletter; it will appear in the next issue.

Thanks for your time and patience!

Micro/RSX available, Pascal-2 compatible

Digital reports that Micro/RSX is now available on the Micro/PDP-11. The Micro/PDP-11 uses the PDP-11/23-PLUS central processor and executes the full set of PDP-11 instructions. A customer tells us that Pascal-2 runs on this system, but you must have the "Advanced Programming Package" and select "NOANSLIB" instead of "SYSLIB" as your system library.

Continued from Page 14

Oregon Software's new Distributor Sales Representative is **Linda Lausmann**. She is the main liason with international distributors, processing sales orders, solving problems, and answering correspondence. In addition to her B.A. in business administration, Linda is experienced in both accounting and sales. **Marilyn Crossgrove**, her predecessor, is moving to England.

Oregon Software's staff is not static. As a company, we try to facilitate the personal and professional growth of each member by encouraging many horizontal and vertical career paths. Since our last newsletter, many have taken new paths.

Collins Hemingway was the original technical writer at OSI and over a period of two years, he established the Technical Publications group. Collins is now the public relations liaison between the company and trade journals, placing technical articles and coordinating informational tours.

Terry Juve has moved from distributor sales to the technical group. As a programmer trainee, she's processing trouble reports and testing software.

Diane Likens has been promoted to Customer Service Representative. She controls inventory for Oregon Software headquarters and the European Warehouse. In addition, she handles special manual orders and special projects, such as improving our record keeping procedures.

Betsy Slonaker, OSI's word processing operator, is now Production Assistant for Technical Publications. She'll be getting manuals and newsletters ready for printing and training in computer operations and the TEX system.

Laura Ronck has been promoted to Programmer. She performs validation tests for new Pascal–2 releases for DEC and Motorola systems; she also sets up release mechanisms.

Senior Software Engineer **Steve Poulsen** now serves as technical consultant to Sales/Marketing. He provides answers to customers' questions about the software and has been working closely with Mary Erichsen on OEM sales. Steve is also a frequent contributor to this newsletter.

As Vice President of Software Development, **Bob Phillips** is hiring new staff and reorganizing OSI's development and support group to prepare for future expansion.

As Purchasing Agent, **Katie Wilding** negotiates with all vendors and handles travel arrangements. She still fills in as receptionist occasionally.

Sandy Profeta has been promoted to Systems Manager and has the full responsibility for maintaining OSI's three DEC systems. She also negotiates purchases of magnetic media for product distribution and new hardware for OSI's computer system.

Pascal
NEWSLETTER
OREGON SOFTWARE

2340 SW Canyon Road
Portland, Oregon 97201

Pascal
NEWSLETTER
NUMBER 9 OREGON SOFTWARE FALL 1984

Pascal-2 spans DEC line with VMS release

Oregon Software's highly optimizing Pascal–2 compiler is now available for the VMS operating system on the VAX and MicroVAX.

Release of the VMS native compiler makes Pascal–2 the only Pascal compiler that provides a uniform development environment across the full line of Digital systems from the Pro 350 to the VAX.

Unlike other language compilers available on DEC systems, all versions of Oregon Software's Pascal–2 provide identical language features, including the same interface with the operating system and hardware and the same set of language extensions. All versions contain identical programming development tools, including an interactive, high-level debugger.

Pascal–2 is integrated into the VAX/VMS system. The compiler allows calls to separately compiled routines written in Pascal, FORTRAN, or MACRO, giving developers access to existing software libraries.

In addition to DEC systems, Pascal–2 runs on M68000-based computers under the UNIX and VERSAdos operating systems.

For current Pascal–2 users, this means that applications code may be moved quickly and easily among many popular operating systems and that development work among these systems may be done with a uniform set of tools.

Tools aid developers

Like Oregon Software's other compiler systems, Pascal–2 for VMS is designed to minimize the time users spend correcting errors and developing products.

Pascal–2/VMS includes the standard kit of software tools provided with all Pascal–2 systems. These include the interactive debugger, program and text formatters; cross-referencers for program identifiers and procedures; a dynamic string package written in standard Pascal; and a set of definitions to allow Pascal-like use of MACRO code.

Compile-time error checking ensures conformance of data types to the ISO standard, locates many uninitialized variables, and detects nearly 150 Pascal syntax errors. Comprehensive run-time checking detects array index errors, illegal subrange assignments, nil pointer references, non-existent case labels, and I/O and arithmetic errors.

In addition to generating error messages for run-time errors, the compiler produces a trace, or walkback, of the source statements leading to the error. Users are able to recover from normally fatal I/O errors and even to write their own I/O error-recovery code. Future releases of the Pascal–2/VMS compiler will provide the execution profiler and the capability to customize error messages for fatal run-time errors, which are already included in our other Pascal–2 systems. Run-time checking may be disabled for maximum performance.

Pascal–2 supports standard, optimizes code

Pascal–2 for the VAX supports all features of standard Pascal, including packed data structures and set types of up to 256 elements. Pascal–2 supports integer types of 1 to 32 bits. Pascal–2 adheres to Level 1 of the international standard (ISO 7185), which includes conformant array parameters to provide dynamic arrays.

The Pascal–2 compiler performs eight types of code optimization designed to generate fast, small code: global register allocation, common subexpression elimination, expression targeting, array index simplification, range tracking, constant folding, dead code elimination, and short-circuit evaluation.

A compilation switch disables the compiler's extended language features, allowing only ISO standard Pascal programs to be compiled. The "standard" switch generates error messages giving the location of all non-standard code, simplifying the conversion of non-standard Pascal programs to Pascal–2.

In this issue...

Pascal–2 for VAX/VMS released . . . . . Page 1
Oregon Software changes . . . . . Page 2
Information exchange . . . . . Page 3
Who to call . . . . . Page 3
Sharing memory on RSX . . . . . Page 4
More expansion . . . . . Page 7
Errors, additions to manuals . . . . . Page 8
The Log . . . . . Page 11
Known bugs . . . . . Page 14
Modula-2 Questionnaire . . . . . Page 16
OPUS Directory . . . . . Page 17

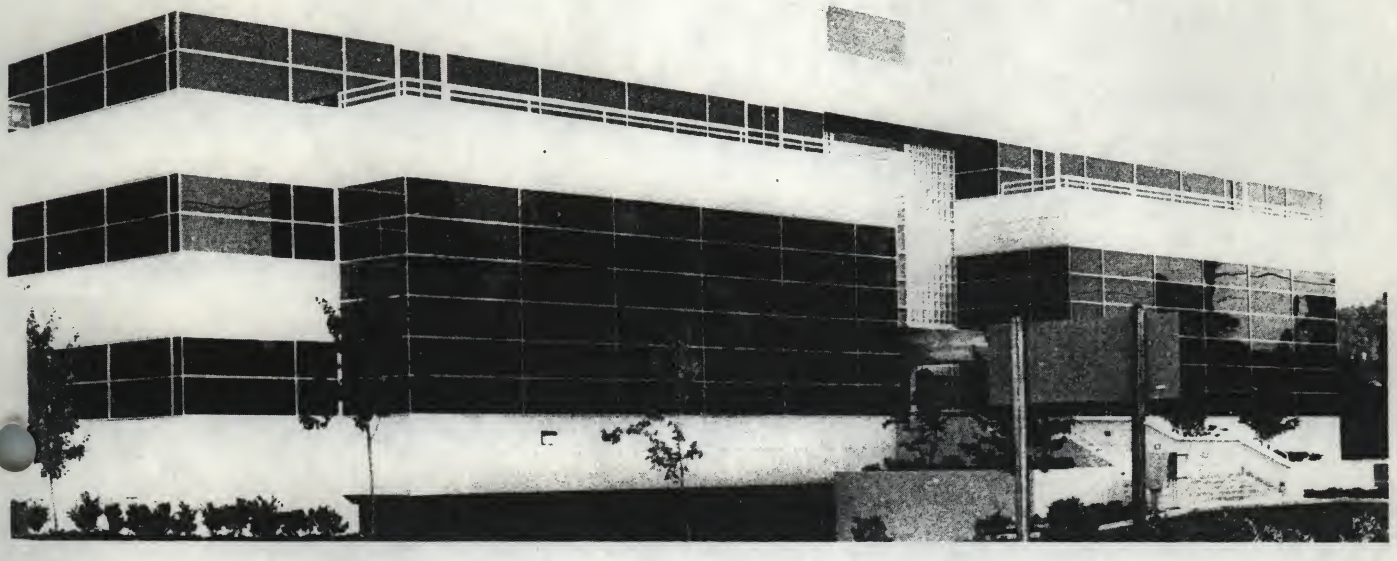


photo by David Spencer

New location – The entire second floor is devoted to staff offices, meeting rooms, a technical library, and the computer room. The employees' cafeteria, a large meeting room, and a large storage room occupy about a third of the bottom floor.

Oregon Software begins new phase

A lot has happened at Oregon Software in the past few months. Perhaps the most exciting event for our staff has been our move to new offices on September 10th. We enjoyed our old building on Canyon Road for its casual atmosphere and for its location, near downtown and next to one of the largest city parks in the country. We did not enjoy the cramped conditions with 50 people crammed into our three separate buildings with a total of 9,000 square feet. Our new location is still near downtown, across the street from a very nice riverfront park, and the building gives us 15,500 square feet with plenty of room to grow.

We passed a major milestone in August when we placed Pascal–2 for VAX/VMS in field test. We are pleased to announce that this product will be available for delivery as of October 22, 1984. Most PDP-11 or M68000 Pascal–2 programs can now be recompiled to run in VAX native mode with little or no change.

We've heard from a number of our best customers that our product support has become inadequate. This news is very disturbing, and we are taking some definite steps to solve the problem. There are two areas of concern: bug-fixing and phone support.

We are working very hard to catch up on the backlog of trouble reports that has built up over the past year, especially for our cross-development software from the VAX to the Motorola M68000. We have hired some new programmers with compiler experience who are working hard to learn Pascal–2 internals and to get problems fixed. The "old hands" on staff are giving them lots of help. For all products, we will provide maintenance releases twice a year, on a set schedule, as we do for our PDP-11 products.

We are also concentrating on teaching our phone support people more about our products and the programming environments in which they are used. Many of these staff members are relatively new to Oregon Software, and they are eager to learn more of the programming folklore that goes with our products.

We are confident that these steps will improve the quality of our technical support and we are grateful to our customers for their feedback on the level of service we provide.

Don R Baccus

Don Baccus, President

Come see us at Booth 2836

Oregon Software will be exhibiting products at DEXPO West, held at the Disneyland Hotel in Anaheim, California, December 11 - 14. Our sales staff will demonstrate new products, including Pascal–2 for VAX/VMS. Technical staff will answer your questions about existing products. We invite all our customers to visit us.

Information exchange

If you need information on technical applications involving Pascal, or if you have an application that might interest other users, send us a brief description for inclusion in the Information Exchange. Your description should follow the format of the items below. Interested parties can contact one another directly.

Two software tools help programmers write correct, portable Pascal programs and help compiler writers produce accurate, bug-free, fully conformant standard Pascal compilers. The Pascal Validation Suite Quality Control Package (PVS) consists of 734 test programs which systematically exercise a Pascal compiler to determine its ability to process programs written in ISO standard Pascal. The Standard Pascal Model Implementation (SPMI) consists of a compiler and interpreter, used in combination to process the validation suite tests correctly, and a static checker which audits Pascal programs for conformity to the ISO standard. The entire SPMI implementation is written in Pascal. The PVS and SPMI are available in machine readable source code on tape and diskette. For more information, contact: Donna Kish, Software Consulting Services, Ben Franklin Technology Center 125, Murray H. Goodman Campus, Lehigh University, Bethlehem PA 18015, (215) 861-7920.

PRM-11 is a Pascal Record Management system for use with Pascal–1 or Pascal–2 under RSX-11M V4.0. PRM-11 also runs under VAX/VMS V3.1 in compatibility mode, with the restriction that shared files may not be opened with `write` access. Documentation is included. Contact: Doug Bliss, Toledo Scale, 1150 Dearborn Drive, Columbus OH 43229, (614) 438-4877.

OPUS Communiqué

Oregon Pascal Users Society

The column for this issue consists of the OPUS membership list for September 1984. We've arranged the pages at the back of the newsletter so that you can cut them along the inside edge, staple them in the middle, and make a 5.5 by 8 inch booklet. Further information will be in our winter issue.

Who to call...

To be connected with the "right" person when you call Oregon Software, you can ask for one of the following people by name or tell the receptionist the type of information you want.

Technical matters – tell the receptionist that you need technical help. She will connect you with the support person for that day.

Updates or support status – ask for Customer Service Manager, Claire Lematta, or explain to the receptionist that you are calling about support renewal.

Manual orders – direct your request to Diane Likens.

Purchases and product information – ask for Sales or tell the receptionist what information you wish. Sales representatives Tim McMenamin, Penny Green, Dan Kane, and Lorie Griffith serve prospects and customers within the United States.

International distributors – For sales inquiries from outside the U.S., ask to speak to Linda Lausman.

Domestic distributors and OEM accounts – ask for Mary Erichsen, our Sales Manager.

Pascal NEWSLETTER
OREGON SOFTWARE
6915 SW Macadam Avenue
Portland, Oregon 97219

Editor David Spencer
Staff Writers
Thomas E. Hanrahan, Mary Hutton, Louise Waitt
Production Assistants
Betsy Slonaker, Jennifer Mulder

The Pascal Newsletter is published quarterly by Oregon Software, Inc., 6915 SW Macadam Avenue, Portland, OR 97219; (503) 245-2202. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

Pascal tasks share memory under RSX

by Steve Poulsen

Even if your computer has a megabyte of memory, and even if your DEC sales representative says otherwise, no PDP-11 program accesses more than 64K bytes of memory at a time. The limitation is inherent in the PDP-11 architecture: Since addresses are 16 bits long, a single program can access only 65536 (64K) bytes or 32768 (32K) words of memory.

With large software projects, this may be a major inconvenience, because programmers must split up any large program into several smaller parts. The use of active page registers (APRs) to share memory between Pascal tasks on the RSX operating system allows these smaller parts to communicate.

First, some background. PDP-11 hardware registers are 16 bits wide, and instruction lengths are multiples of 16 bits. More important, memory addresses are stored as 16-bit values. On most PDP-11 computers, the memory management hardware is used to map virtual addresses to physical memory addresses. Virtual addresses, which are the 16-bit addresses used by a single program, are always in the range of 0 to 65535. For each virtual address, the memory management hardware computes an 18-bit or 22-bit physical address. Physical addresses may range from 0 to several million. The allocation of physical memory to tasks is a primary responsibility of the operating system.

You can use the APR in one task to map to the same physical addresses as an APR in another task, which allows the two tasks to share memory (data). This requires four steps, described in detail below: 1) create a special portion of physical memory called a partition to hold the data to be shared; 2) create a MACRO file that allows the Task Builder to map tasks to that partition; 3) install the task image for the partition; 4) use `loophole` to force Pascal variables to the start of the partition.

Before creating a partition, you must understand the allocation of virtual memory within a single task, which is controlled by 8 APRs. Each APR controls up to 8K bytes of memory. The following table shows the address ranges controlled by each APR.

The memory management hardware uses the upper three bits of the 16-bit virtual address to determine which APR (0 to 7) to use to map the physical address. The remaining 13 bits specify an offset in the range from 0 to 8191 bytes from the physical address mapped by the APR. Each APR has a length and an access mode associated with it. The length specifies the number of 64-byte blocks mapped by the register, and the access mode describes the kind of references that can be made (read-write, read-only, or no access).

Address Ranges Controlled by APRs

Octal Addresses	APR
000000-017777	0
020000-037777	1
040000-057777	2
060000-077777	3
100000-117777	4
120000-137777	5
140000-157777	6
160000-177777	7

Consider a program containing 30000 (octal) bytes or 6K words. For this task, the operating system sets APR 0 to map virtual addresses from 0 to 017777, its limit, and APR 1 is set to map addresses from 020000 to 027777, to make a total of 30000 bytes. Both APRs are set with read-write access automatically by the system. APRs 2 through 7 would be set to "no access." If the program attempts to access a virtual address in the range 040000 to 177777, a "memory protection violation" is generated. The allocation of virtual memory can be controlled by the Task Builder or by special system services called at run-time. A Pascal program allocates the heap and stack by asking the operating system to allocate additional virtual memory to the task as needed.

As an example, assume that you want to share 80 bytes of data between two Pascal programs. In particular, you want to write one program to leave a message in the shared memory area and a second one to retrieve and print the message.

You need to create a partition to hold the data. The partition specifies which area of physical memory is to be shared and prevents the operating system from using the memory for other purposes. Your system manager can create the partition dynamically or with VMR, so that the partition is installed each time the system is booted. In either case the commands are similar.

Your system is probably already making full use of available physical memory. To add a new partition, you need to reduce the size of an existing partition. This can be done with the SET/TOP command. The GEN partition is usually the largest partition, and it can usually be reduced in size without adverse side-effects. Use the PAR command to view the current partition allocation.

You want to create a common partition. Note that the GEN partition in Figure 1-A holds 515700 bytes and occupies physical memory in the range of 242100 to 760000 (760000 = 242100 + 515700). You want to allocate 80 (decimal) bytes of data. Since partitions are always multiples of 64 bytes, round the number needed up to 128 bytes (or 200 bytes in octal). To make room for 200 bytes at the end of the GEN partition use the command:

```
>SET /TOP=GEN:-2
```

The -2 parameter means that the top of the GEN partition is reduced by 200 bytes. (Memory addresses and sizes are in multiples of 100 octal bytes, so the final two zeroes are dropped.) By reducing the size of the GEN partition, you leave room for your common area DATA in the range 757600 to 760000 (757600 = 760000 - 200). To create the DATA common area, use the command:

```
>SET /MAIN=DATA:7576:2:COM
```

This creates a common partition called DATA based at 757600 with a length of 200 (octal) bytes. Use the PAR command to see the new partition (see Figure 1-B).

If you use VMR to create the DATA partition, reboot your system to make the partition available.

MACRO file describes data

After creating the DATA partition, you must next create a MACRO file to specify the amount of data to be placed in the partition. This enables the Task Builder to map tasks to the partition. (The size of the data need not be the same size as the partition, though the partition must always be at least as large as the data. This is why both the partition and the MACRO program are needed. In the next example, the partition size is the same size as the data, but you could—wastefully—put a small amount of data in a large partition.)

To share the 80 bytes of data, create a MACRO source file called DATA.MAC that allocates 80 bytes of data. The .BLKB directive allocates 80 bytes of data.

```
.TITLE  DATA
.BLKB 80.

.END
```

The decimal point after the 80 is required to interpret the number as a decimal number rather than an octal number. This file is assembled to produce DATA.OBJ.

```
>MAC DATA=DATA
```

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 515700 MAIN SYS
       041600 242100 020000 SUB  (...MCR)
```

A. Existing Partition

```
>PAR
EXCOM1 067734 070000 014700 MAIN COM
EXCOM2 067670 104700 010200 MAIN COM
LDRPAR 067624 115100 002600 MAIN TASK
TTPAR  067260 117700 030000 MAIN TASK
DRVPAR 066734 147700 003700 MAIN SYS
       066670 147700 002300 SUB  DRIVER -DL:
       066570 152200 001400 SUB  DRIVER -DX:
SYSPAR 066470 153600 010100 MAIN TASK
FCSRES 066424 163700 032000 MAIN COM
FCPPAR 066360 215700 024200 MAIN SYS
       041204 215700 024200 SUB  (F11ACP)
GEN    066314 242100 515500 MAIN SYS
       041600 242100 020000 SUB  (...MCR)
DATA   041430 757600 000200 MAIN COM
```

B. New Partition

Figure 1. Partition Allocation

The first column is the name of the partition. The second column is the internal location of the partition control block within the monitor and is not too useful. The third column is the base address of each partition or subpartition. The fourth column is the length of the partition. The fifth column is the type of the partition. MAIN is the main partition while SUB is a subpartition of a MAIN partition. The sixth column shows that a partition can be a common area (COM), a system-controlled partition (SYS), a task-controlled partition, a device driver (DRIVER), or a device common (DEV).

Installing the task image

The Task Builder creates the task image and symbol table for the partition. Several special options must be used. The /-HD option creates a task image without a task header, and the STACK=0 option prevents the allocation of stack space in the task. The PAR option gives the name of the partition in which the task will be installed (DATA), the virtual base address of the partition (160000, the base address for APR 7) and the size of the partition (rounded up to 200 bytes). All numbers are expressed in octal. APR 7 is used to map the shared data because it is the highest available APR (see table). Since Pascal tasks may use additional APRs for the heap as they expand, it is best to use APRs at the high end of virtual memory. This provides the maximum memory area for the program.

You must use Task Builder commands to create three files: DATA.TSK, the task image for the partition; DATA.MAP, the map file; and DATA.SYM, the symbol table, which describes the partition and its contents. The symbol table is used by the Task Builder when Pascal programs are linked with the shared data area, as explained further below.

```
>TKB
TKB>DATA/-HD,DATA,DATA=DATA
TKB>/
ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=DATA:160000:200
TKB>//
```

DATA.TSK should be installed with the INS command:

```
>INS DATA
```

'Loophole' forces data to virtual memory

You can now write Pascal programs that share the data. To specify which data is to be shared, you must force the Pascal variables to appear at virtual location 160000 (octal). Any data at that virtual location is mapped with APR 7. Then use a special Task Builder option to cause APR 7 to map to the DATA partition you created above.

Using Pascal-2, you can force data to be referenced at a particular virtual address by creating a pointer to the data. Then assign the pointer a virtual address with Pascal-2's **loophole** function.

Sample program WRITE shows how to force variables to appear at virtual location 16000 with the **loophole** function. The **type** statement does not allocate any memory for the **message** you want to share. It only describes what a message looks like. The **type message_pointer** is a pointer to a message. The only data allocated to the program is p which is a pointer to a message. In the first statement of the program, the **loophole** function converts the octal integer 160000 to a **message_pointer** which can then be assigned to the pointer p. The second statement of the program prompts you to type a message and the third statement reads the message. Since the variable **p** is a pointer to a **message**, p^ is the message itself. Note that the storage for the **message** is not contained in the program itself, but is allocated in the small MACRO program installed in the DATA partition.

```
program WriteMessage;
  type
    message = packed array [1..80] of char;
    message_pointer = ^message;
  var
    p: message_pointer;
begin
  p:=loophole(message_pointer,160000B);
  write('Type a message:');
      readln(p^);
end.
```

Compile the program WRITE.PAS:

```
>PAS WRITE
>TKB
TKB>WRITE/FP/CP,WRITE=WRITE,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>RESCOM=DATA/RW
TKB>//
```

The RESCOM option specifies the location of the symbol table file, DATA.SYM, which describes the shared data area DATA. The /RW switch requests both read and write access to the common data area. The /RO switch can be used to request read-only access. If you copy DATA.TSK and DATA.SYM to LB:[1,1], then instead of RESCOM, use the COMMON option: **COMMON=DATA:RW**.

You can write a similar program to read the message stored in the common data area.

```
program ReadMessage;
  type
    message = packed array [1..80] of char;
    message_pointer = ^message;
  var
    p: message_pointer;
begin
  p:=loophole(message_pointer,160000B);
  writeln('Message: ',p^);
end.
```

Here, the pointer p is initialized as in the first program. The second statement prints out the message pointed to by p. Compile and build this program (READ.PAS) as follows:

```
>PAS READ
>TKB
TKB>READ/FP/CP,READ=READ,LB:[1,1]PASLIB/LB
TKB>/
ENTER OPTIONS:
TKB>RESCOM=DATA/RO
TKB>//
```

The RESCOM option causes the Task Builder to read the file DATA.SYM to obtain information about the shared common area called DATA.

Now you can use the WRITE program to write a message to the shared data area, and the READ program to read back the information.

```
>RUN WRITE
Type a message: this is a test

>RUN READ
Message: this is a test

>RUN WRITE
Type a message: so is this

>RUN READ
Message: so is this

>RUN READ
Message: so is this
```

Setting up a partition for the sharing of data and using `loophole` to reference it allows individual programs to access more than 32K words of *total* data, even though no more than 32K words of data may be accessed at any one instant. In practical terms, the maximum additional amount that can be shared is 24K words, because even the smallest Pascal program on RSX takes up about 8K words of memory.

New staff adds to Oregon Software mix

Assistant Systems Manager, **Jim Anderson** joins Oregon Software after 17 years with Boeing Computer Services. In addition to his job, which Jim describes as "beautiful," he enjoys photography, playing guitar and banjo, writing poetry and working on hydroplanes and drag racers with his brother.

David Barnes joined the Technical Publications group this September. A graduate of Ohio State in communications theory, Dave has been a technical writer at a large company in Columbus for five years. He enjoys learning computer languages and has experience writing programs in Macro-10, BASIC, FORTRAN, PL/1, and BLISS. Dave claims "only a reading knowledge of Pascal and C" and looks forward to Pascal programming at Oregon Software. Living in the Pacific Northwest is a new experience for the Barnes family: "I'll have to learn to grow roses," Dave says.

Accounting Clerk, **Julie Berrigan** performs many tasks in our accounting department, especially processing and collecting accounts receivable. Her one-year-old, Matthew, keeps Julie running when she isn't at Oregon Software.

JoAnn Bertram became Department Secretary for Sales and Marketing in July. She claims to be the only University of Washington graduate who took ten years to get a BA in English literature; she was also putting her husband through school and taking care of two children at the time. Currently, JoAnn and her husband are remodeling their house.

photo by David Spencer

Voices of Oregon Software – Katie Wilding (left) has trained Rita Gorham (center) to take her place as receptionist. Sharon Lodewick (seated) serves as back-up on the phones.

Denise Bristow is improving customer support by organizing all trouble reports by product. She received a degree from Portland Community College in applications programming last March and joined Oregon Software in August. Denise has travelled extensively in Europe, North Africa and Hawaii. She likes jogging, rafting, hiking, mountain climbing and scuba diving. She hopes to take up raquetball again now that she's out of school.

Continued on Page 10

Errors, additions to manuals

The Technical Writing staff keeps a list of changes and additions to be included in each manual. We receive information from readers through Documentation Evaluation Reports (the DER forms at the back of each manual), Trouble Reports, and support calls. Every six months, usually coinciding with a release of the software, we issue an update package.

The cycle of update packages is staggered. During the interim between updates, we list changes and additions in this newsletter and in the Release Notes.

All Pascal-2 manuals

We've found the following errors in *Pascal–2 User Manuals*, Version 2.1 for RSX, RSTS/E, RT-11, and UNIX.

A bug in a Debugger example

The description of the use of negative numbers as *count* parameters for the L command is incorrect. A negative *count* parameter causes the Debugger to list statements up to and including the statement number indicated in the command. The example should read:

```
} L(Rotate,4,-2)
  16    3        A[I] := A[I+1];
  17    4      A[Last] := A[First];
```

Language Specification, extended-range arithmetic

Substitute the following passage for the existing third and fourth sentences of the paragraph:

> Normal arithmetic operations, with the exception of division and modulo, are performed on extended-range variables. Comparisons and division are signed.

Pascal-2 RSX manual

The *Pascal–2 User Manual, V2.1 for RSX*, contains the following errors.

Overlays, page 2-19

In the TKB multiline command example, delete the third line; the /MP switch eliminates the need for this entry.

Support library, page 2-21

Under "Support Library Data Definitions," the words "The file" (fourth line, first paragraph) should say "LIBDEF" to clarify which file defines the file name block.

Multiple source files, page 2-53

In the source file EXAMPL.PAS the first **%include** directive (**%include 'config'**) should be deleted so the command at the bottom of the page does not include the file twice.

Location of the compiler's work files, page 2-54

In the fourth paragraph, the "**time**" compilation switch should be "**times**," for completeness.

Using event flags, page 2-67

The example program DELAY fails to take into account that an enumerated type with fewer than 256 members is allocated one byte. Potential problems arise when the enumerated parameter is passed to the **nonpascal** procedure MARK because MARK actually requires that a two-byte (word) parameter be passed. Defining the individual time intervals to be constant integers takes care of the problem because the size allocation for integers is two bytes. In the example, change:

```
type
  TimeInterval = (Ticks, Seconds,
                  Minutes, Hours)
```

to read:

```
const
  Ticks = 1;
  Seconds = 2;
  Minutes = 3;
  Hours = 4;
```

Margins, page 5-48

In the third paragraph, the default value of the left margin (L10) should be L0. The value in the table immediately preceding that paragraph is correct.

Pascal Procedures in Resident Library, pages 2-77,78

On each page, the number 5353 octal should be 5354.

Existing Procedures in a Resident Library, page 2-79

The first full line of the task build command at the top of the page should read:

```
TKB>WRTSUM/FP/CP,WRTSUM=WRTSUM,LB:
    [1,1]PASLIB/LB:$FCSJT,LB:[1,1]PASLIB/LB
```

'Origin' Declaration, page 3-20

The reference to the section "Size Function" should be referencing the section "Size and Bitsize Function."

Debugger Guide, pages 4-4 to 4-19

When beginning a debugging session, the 2.1B Debugger identifies the program being debugged in a slightly different

way than the 2.1A Debugger. Examples on these pages should show the corrected line: **Debugging program ROTAT**.

Example, page 5-34

In the **2$**: block of the MACRO-11 example, the variable **r0** in the comment at the end of the **bne** statement should be **n**.

In the last paragraph, the word **restore** should be **rsave**.

Pascal-2 RSTS/E manual

The *Pascal–2 User Manual, V2.1 for RSTS/E*, contains the following errors.

Implementation of escape differs for RSTS

RSTS interprets the standard **escape** character **chr(27)** to be a line-terminator and returns a **$** prompt. To get a true escape character, you must set the high-order bit by using **chr(155)**.

Example: function 'NewOK', page 2-35

In the listing of **NewOK**, delete the reference to **$$HEAP** in the comment for the **space** function definition.

Error termination Status, page 2-50

In the second sentence of the note at the end of this section, the words "It is" should be inserted before the word "provided."

Pascal-2 RT-11

The *Pascal–2 User Manual, Version 2.1 for RT-11*, contains the following errors.

Copyright, page ii

The trademark entries should include "TSX-Plus is a trademark of S &H Computer Systems, Inc."

Example: function 'NewOK', page 2-30

In the listing of **NewOK**, delete the reference to **$$HEAP** in the comment for the **space** function definition.

Multiple source files, page 2-44

In the fifth paragraph, first sentence, the second "are" should be deleted.

EXITST walkback for 'severe error' status (4)

The **Exitst** procedure is a support library routine that sets the termination status of a program and stops the program when a "severe error" status is detected. The procedure's integer argument determines the termination status for any program that calls it. When a "severe error" status of 4 is passed, the procedure also invokes the post-mortem analyzer to create a walkback of the program execution from the point of failure.

Pascal-2 UNIX manual

The *Pascal–2 User Manual, V2.1 for UNIX*, contains this error:

Storage allocation for packed record, page 2-17.

For 68000 users, the last line should read "beginning at bit 8 (most significant bit)."

All Pascal-1 manuals

With the release of V1.3D, we'll issue a new user manual. In the meantime, the V1.2K manual and a supplement are the documentation.

Run-time checking switch is renamed

Oregon Software provides a set of notes entitled *Conversion From Pascal–1 to Pascal–2* for those customers upgrading from Pascal–1 V1.2 to Pascal–2 V2.1. If you have this document, please make this change. On page 1-7, you are told to change the embedded directive **$A** to **$arraycheck**. The correct change is to **$indexcheck**.

Addition to supplement

Users should note that the *Supplement to Existing Manuals* for Version 1.3 does not describe a newly added feature: Pascal–1 Version 1.3 now accepts **$** in identifiers.

Pascal-2 VERSAdos V2.0 manual

The following items should be added to *Pascal–2 Version 2.0/M68000 User Manual.*

Debugger, page 126

The description of the use of negative numbers as *count* parameters for the **L** command is incorrect. A negative *count* parameter causes the Debugger to list statements up to and including the statement number indicated in the command. The example should read:

```
} L(Rotate,4,-2)
  16    3        A[I] := A[I+1];
  17    4      A[Last] := A[First];
```

Error termination status, page 37

The following text should be added to the section on "Run-Time Error Reporting."

Both the Pascal–2 compiler and Pascal programs return a termination status when they exit. The Pascal–2 compiler terminates with a "severe error" status if it detects

compilation errors. Upon detecting an error while running, such as "subscript out of bounds," a Pascal program also terminates with a "severe error" status. Otherwise, a "successful completion" status is returned.

The Pascal support library contains a routine that may be called from Pascal programs to set the termination status and stop the program. To use this feature, declare an external procedure named **exitst**. This procedure, defined in the support library, takes an integer argument, as shown:

```
procedure Exitst(Status: integer);
{ procedure declaration }
  external;
```

Call the procedure at a point in the program where you want to exit in case of a severe error, as shown:

```
begin       { program Severe }
  :
  Exitst(4);    terminate with severe status
  :
end.        { program Severe }
```

A status of 1 means normal termination; any other status means that an error terminated the program.

Compilation Switch /longlib

In order to take advantage of "short references and instructions" which use 16-bit addresses, the Pascal–2 support library under VERSAdos resides in the low end (first 32K bytes) of memory. Support library subroutines use short references internally and the compiler generates calls to the support library using the 16-bit addresses.

The new **/longlib** switch causes the compiler to generate full 32-bit address calls to the support library. Users who wish to use **/longlib** must change all internal use of short instructions in the library's source code before relocating the support library.

Concurrent Programming manual

Users of V1.0A should make this change.

Booting Instructions, page 4-10

You are told to use the supplied program MAKEBOOT to convert the load module to a bootable image. Users must also compile and assemble FHSCAL.SA and FHSUTI.PA, two modules that are shipped with the utilities. These modules are then linked with MAKEBOOT.

New staff

From Page 7

Lyn Gabel, Administrative Assistant to our President, has been writing a company personnel manual and will be doing research projects. A graduate of Oral Roberts University with a BA in psychology, Lyn says "I was a Marine brat, so we lived all over the U.S. when I was growing up." Lyn and her husband are Portland Winter Hawks boosters; they board one of the team's defensive players, a student from Canada, during the hockey season. Fortunately, Lyn "loves to cook."

Rita Gorham came from 4A's Temporary Service to fill in as our receptionist. She did such a good job that we asked her to stay. Rita enjoys cross-country bicycle touring and downhill skiing. She hopes to take up scuba diving so she can add seashells to her collection, preferably some from the Barrier Reef area of Australia.

Formerly an intern programmer, **Bruce Graham** became a full-time programmer in August. A graduate of Reed College in mathematics with additional coursework in computer science at OSU, Bruce has been working on the UNIX debugger. Before joining Oregon Software, Bruce spent five years in Alaska as a fire fighter and taught high school math and physics in New Zealand. He enjoys woodworking, particularly building boats.

Technical writer **Mary Hutton** joined Oregon Software in July and immediately began revising manuals for new releases of the software. A graduate of both Stanford (BA in Latin) and the University of Washington (BS in scientific and technical communication), her interests include archaeology, mathematics, language study, anthropology, and all types of crafts.

Sales representative **Dan Kane**, a Portland native, recently graduated from Oregon State University (BS degree in Business Administration) where he concentrated on marketing and sales management. Dan says he likes sales because it involves communicating with people. His avocations are playing as much golf as possible, playing softball, and skiing.

Krzysztof Krüger, a native of Poland, comes to Oregon Software via Aachen, Germany, where he programmed in Pascal-2 for three years. In addition to Polish and German, Kris is fluent in Russian, French, and English. After he gets his family settled, he hopes to resume his favorite hobbies, basketball and photography. What does he think of his new home? "I'm glad to be here," says Kris.

Continued on Page 15

The Log: errors, work-arounds, changes

As in previous issues, this log also describes the significant changes in Pascal-1, Pascal-2, and SourceTools. As a service to users who may not have the current release, we supply work-arounds where possible.

For the first time, we've listed many known bugs: those problems reported by users and verified as bugs by Oregon Software. By conveying this information, we spare you the labor of tracing and reporting bugs unnecessarily and give you some indication of a particular report's status.

To use this log, you must know the release level of your software. (The release number is printed on the headline of all program listings.) Then review the log to determine the changes made since the release of your version. As an additional reference, we've placed the Trouble Report numbers in square brackets at the end of each log entry.

If the changes are of particular importance to your application, your Designated Contact Person should request an update, in writing. Upon receipt of your written request, you will receive the latest version of the software.

Pascal-1

Version 1.3C has just been released. We have not received any Trouble Reports from users.

Pascal-2

Version 2.1D corrects a number of problems common to PDP-11s with RSTS/E, RT-11, or RSX-11 operating systems.

Packed boolean array problems

The compiler no longer generates incorrect code when a packed array of boolean crosses a byte boundary. Under V2.1C, only the lower byte of the word was fetched so bits representing the higher elements of the array were not set properly [1173, 1319, 1986].

Declared as a **var** parameter, the packed array of boolean is now passed to a procedure correctly [1269].

Record field as a parameter

The compiler no longer generates bad code when certain nested record fields are passed as a parameter [1191].

Negative stack offset for common subexpressions

Depending upon the location of a variable's declaration, two identical calculations involving certain functions, including sine and cosine, could have different results. The library routines for the functions no longer cause the compiler to reference a negative offset from the stack pointe [1572].

Wrong results from integer comparison

A comparison between an integer and a subrange that is a element of a packed record no longer produces an incorrec result [1572].

Incorrectly addressed real variable

Certain globally allocated real variables are now addresse correctly in large programs [1613].

Dynamic string package errors

The `Insert` routine no longer prints the error messag `Array subscript out of range` if the combined lengt of the two input strings exceeds the target string's length Instead, the insert string is concatenated onto the end c the target string and truncated [1782].

The `Concatenate` routine no longer reports errors whe the string to be concatenated overflows the target string The input string is truncated to fit.

%Page directive doesn't work

On listings, **%page** now increments the page number cor rectly [1914].

Reserved instruction trap error

Using a **packed record** containing integers in a neste procedure call no longer causes a reserved instruction tra [1988].

Utilities don't compile

With Versions 2.1A and 2.1B, attempts to compile th utilities resulted in an **Unknown Pascal run-time erro** [2027, 2303a, 2393].

Illegal instruction gives "compiler writer error"

Using certain illegal instructions resulted in the error mes sage for internal problems instead of the appropriate erro message [2042].

Sets generate bad code

Set constructor expressions of the form **[var1..var2]** n longer generate bad code at times [2046, 2046a].

No error message

The compiler now gives an error message for attempt to pass an element of a **packed** structure as a variabl parameter [2151].

Incorrect use of 'R0' for procedure calls

The compiler now saves the contents of the **R0** register before a call to a **nonpascal** procedure [2177a].

Set type produces wrong results

The declaration and statement shown below now compile correctly [2177b].

```
type  a = set of (one,two,three);
var   b:a;

begin
  b:= [one,three];
  write(b * [] <= []);
end.
```

Function returns incorrect result

The compiler now correctly changes bit lengths to byte lengths when a function returns a structured type [2177c].

Bit optimizations incorrect

Compiler optimization of certain user-defined procedure calls no longer produces an incorrect sequence of execution [2184].

PB formatter rejects 'nonpascal'

The formatter now recognizes the **nonpascal** directive and treats it exactly like the **external** directive [2202].

String comparison range wrong

For string comparisons, **ord(char)** is now in the range 0..255 instead of -128..127 [2248].

Unsigned characters treated as signed

The compiler no longer treats 8-bit characters as if they are signed numbers at times [2329].

Assignment statements cause failure

A complicated series of assignment statements involving arrays of type **real** no longer fails. [2403].

Compiler consistency checks reported

Version 2.1D fixed certain conditions causing the error message **Undeleted temps in procedure...** [1142].

Compatibility mode address trap

The Task Builder does not give a start address to programs compiled without a main program and with the embedded **main** switch. When executed, such programs trap because no program may start at location 0 on the PDP-11. No error message is issued by the Task Builder or by the Pascal–2 compiler, but the code generated is correct [1989].

Changes to RT-11

Misleading error message for nonexistent files

The compiler no longer prints the error message **Unknown Pascal run-time error...** for an attempt to compile a nonexistent file or a file containing a **%include** of a nonexistent file [2235].

Changes to RSX

/apd switch problems

Programs appending records to an existing file do not occasionally crash after crossing a block boundary. Formerly, these programs trapped out to RSX with **T-bit Trap** or **BPT executions** error message and changes in the size of the program caused a different error message [2392].

'Forini' incompatibility

The FORTRAN initialization routine, **forini**, now works properly on VAX in RSX compatibility mode [2370]. (This bug was not fixed in the V2.1C release, as reported in the last issue.)

Memory protection violation

The value of **R0** is now initialized before adding the offset [1273].

Changes to UNIX

Version 2.1E is the current release of the UNIX/68000 compiler. We've corrected a number of problems in the V2.1D release and completed development work on the Debugger.

Debugger

The Pascal–2 Debugger now functions under UNIX.

Unnecessary checking in 'for' statement

The range-tracking optimization no longer performs unnecessary final-limit checks on generated code.

Bad integer error message

Certain cases of incorrect code generated by **packed** fields in records have been corrected.

Trap on exponent underflow

Support library routines now return a signed zero instead of causing a fatal run-time trap.

Floating point division

The entire divisor is now restored to the partial remainder, not just the low-order word of the divisor.

Remainders and quotients inaccurate

The correct register is now accessed for division of operands longer than 16 bits. Previously, quotients could be in error by one bit and remainders (**mod**) could suffer inexpilicable loss of significance.

Inaccurate results of 'If' conditional

The expression '`if x mod y = 0`' no longer produces inaccurate responses.

'For' loop failures

For loops with a control variable that is an **enumerated** type of one byte in size now generate correct code [1918].

Integer constant range checking

Expressions of the form *integer-constant* **in** `[0..255]` no longer fail.

Conformant array parameters

Conformant array parameters in **forward** and **external** procedure declarations no longer take 25 minutes to compile [2311].

Loss of stack pointer

Very large programs no longer crash due to loss of the stack pointer.

Real number accuracy

Nested functions that return a **real** value no longer generate incorrect code occasionally.

'Undeleted temps' error message

Certain **for** loops no longer generate the error message `Undeleted temps in procedure ...` [2020].

Order of declarations

Declaring an **external** function before a global **var** declaration no longer generates an error message.

Bad 'case' statement error message

An untested value within a **case** statement now returns the error message `No case provided for this value` [2469].

Changes to VERSAdos

Version 2.0N is the current release of the VERSAdos native compiler. We've corrected some of problems that were in the initial release (V2.0M) and have V2.0P, containing more bug fixes, in field test. We've skipped an O-release because V2.0O is visually and verbally awkward.

Debugger

The Pascal–2 Debugger now runs under VERSAdos.

External use of conformant array parameters

The compiler no longer rejects external use of conformant array parameters.

File access problem

The compiler can now access files in directories other than the current one.

Origin extension

Variables used with the **origin** extension now generate correct code.

Changes to cross-development systems

The Cross-Development System's initial release for RSX-hosted systems was Version 2.0M in early 1983. Version 2.0M for VAX/VMS hosts was released later in the same year.

Version 2.0N corrected the following problems for cross-development systems hosted on both VMS and RSX.

Conformant array index variable

The compiler no longer generates bad code when passing a conformant array to a subroutine as a **var** parameter and attempting to access an element of the conformant array with an index variable that was not declared at the same nesting level as the conformant array parameter. Formerly the wrong level of addressing was used during array index calculation; enabling checking had no effect on the error.

External use of conformant array parameters

The compiler no longer rejects external use of conformant array parameters.

'For' loop with computed limits

A **for** loop with computed limits now generates correct checking code when the index variable is within range.

GLOBAL$$ constant

For modules with a main program and a global data area greater than 32K bytes, the constant now generates correct code.

/NOCHECK switch

Checking can now be turned off; this switch also disables stack overflow checking.

ORD operator

When applied to a function of type **char**, the **ord** operator now generates correct code.

Origin extension

Variables used with the **origin** extension now generate correct code.

Set types error message

The compiler now generates the appropriate error message when encountering a set constant of base type **integer** which lies outside of the 0..255 subrange limit. Previously, it trapped out to the operating system with an access violation.

Structured constant traps

A duplicate definition of a structured constant produces an appropriate error message instead of causing the compiler to trap out to the operating system with an access violation.

Changes to VMS-hosted system

Version 2.0N, for VAX/VMS systems, corrected the following problems.

Multidimensional conformant arrays

The compiler no longer traps out to VMS with a divide by zero exception when compiling a two-dimensional conformant array parameter.

Pack/Unpack

The transfer features **pack** and **unpack** now function correctly when operating on **packed array of integer** subranges that are allocated 2 bytes per element: e.g., **packed array**`[1..5]` **of** `0..1000`.

SourceTools

Version 1.0C is the current release. Newsletter #8 reported the bugs fixed in that release.

Miscellaneous notes

The Pascal Standard does not define the **mod** operation for negative divisors. If either operand of the function is negative, its result may be unexpected. Examples: -30 **mod** 8 yields a result of 2, -23 **mod** 8 yields 1, -6 **mod** 5 yields 4. Programs must check for this condition.

Known bugs

We've not fixed all the trouble reports we've received for Pascal-2. Many bugs have been logged but not verified, which means we cannot categorize or describe them by type of problem. By the next newsletter, we intend to reduce the backlog to the point where this section is more informative.

In the meantime, we're listing those trouble reports that we've verified and scheduled for subsequent releases.

Pascal-2 VERSAdos

The following problems with Version 2.0N have been reported and verified for the native compiler.

MOD function

The **mod** function does not produce an error message for negative integers. (See the explanation under "Miscellaneous Notes")

Statement numbers for input don't match output

The statement number in the assembler output code (.SA extension) does not agree with program statement numbers when files are input with the **%include** switch.

Utilities fail

The utility programs may run out of memory on larger files. Relinking them may fix the problem.

Pascal-2 cross-development systems

The following problems with the V2.0N release have been reported and verified for VMS- and RSX-hosted systems.

Bit test instruction gives illegal address

The compiler generates an illegal addressing mode when using the bit test (BTST) instruction for a construct $a\ IN\ b$ where b is a set expression evaluating to a constant at compile time and a is a variable.

Cross-assembler faulty

The OASYS Cross-Assembler occasionally rejects valid code.

Error checking code

The compiler does not generate checking code for subranges in the form `0..`**maxint**.

Field width error with 'writeln'

At run-time, the expression `writeln(value:n);` prints an indefinite number of blank lines under some conditions, e.g., printing hexadecimal output with the field specification `(value:-8)`.

For loop executes once only

A `for` statement in the form

For *integer*:=*integer* `div` *integer* `to` *integer* `div` *constant*

sets the loop index to the final value, so that the loop is executed once only.

Size limit for arrays

The size limit set for certain declarations of arrays is 8M bytes instead of 16M bytes.

Statement numbers for input don't match output

The statement number in the assembler output code (.SA extension) does not agree with program statement numbers when files are input with the `%include` switch.

Truncated integers

No error message is generated if the value returned when an integer read from the standard input file (`P$4`) must be truncated to fit in the storage allocated for the subrange.

Unacceptable abbreviations

PASMAT accepts only the abbreviated form `/o` for the `/options` switch.

Unacceptable file names

PASMAT does not accept long file-name arguments.

Pascal-2 RSX-hosted system

The following problems with the V2.0N release have been reported and verified.

Characters reversed

The characters of a string constant are reversed in a structured constant containing both strings and numbers.

'MOD' function

The `mod` function does not produce an error message for negative integers. (See the explanation under "Miscellaneous Notes")

New staff From Page 10

Sharon Lodewick spells Rita at our reception desk and is training to be a word processing operator. A native Oregonian, she graduated from Sunset High School last spring and will study American literature in the future, perhaps to become a writer or a teacher. She plans to go to Portland State part-time when she's not answering the Oregon Software phones.

Lois McClain became our Tele-marketing Representative in July after six years at American Data Services as a company liason to financial institutions and two years as office manager for All West Display. She plans to learn more about programming at Oregon Software. An avid football fan, Lois "never misses a game" with her six-year-old son. Like Lynn, Lois comes from a military family and has lived in many western states. She collects Oriental porcelain figurines.

Jennifer Mulder worked during the summer as an Intern in the Technical Publications group. She has gone back to BYU for another year of college, including coursework in computers.

As secretary for the Sales Department, **Julie O'Brien** helps end users, distributors, our customer service and OEM sales staff, by sending out project correspondence and literature. Julie recently graduated from Portland State with a degree in social sciences and plans to do volunteer work at the Metro Crisis Center. A native of Oregon, Julie has three children and likes to play tennis.

Cyndy Smith has joined our Technical Department as secretary. A graduate of the University of Oregon in business management, Cyndy will be "taking care of the administrative details and handling special projects" for our programmers and software engineers. Cyndy just returned from three months touring Europe and two years living in Tsuruma, Japan, where her husband worked as an architect for a military contractor. Cyndy enjoys practicing the flower arranging she studied in Japan.

Modula-2 ?

I'd like you to help us understand the growing interest in Modula-2 so that Oregon Software can better plan "if" and "how" we might approach delivering a Modula-2 product at some point in the future. This won't reduce our commitment to expanding our Pascal product lines and improving customer support.

Please telephone me at our free 800-874-8501 number or send me a photocopy of the following questionnaire with your responses plus any comments that you feel might help our planning. Ask anyone else in your organization with strong views on Modula-2 to respond as well.

1. What's your level of interest in Modula-2? Please write a short description or check the items that apply to you:

( ) a. I'm already using a Modula-2 compiler for _____

system. Provided by _____ Does it meet your needs? _____

_____

( ) b. I plan to acquire a Modula-2 compiler. When?_____

( ) c. I'm interested in Modula-2 but have no specific plans.

( ) d. I'm not particularly interested in Modula-2.

2. What capabilities not in Pascal-2 would cause you to switch to Modula-2, C, Ada, or some other language?

3. If you could get a reliable, efficient Modula-2, would it be your first choice among languages? Rank the languages you would prefer to use.

4. What type of work might you do with Modula-2? Examples: process-control, real-time, teaching, systems programming, applications, utilities, business?

5. Rank the auxiliary packages you most want to see in any new language products? Examples: debuggers, profilers, construct editors.

6. What do you think about the view of some of our customers that a new language is not needed? They suggest that we further enhance our Concurrent Programming Package (CPP) to include more of the functionality of Modula-2.

Name _____ Title/Group _____

Company/Organization _____ License Number _____

Address _____

_____ Telephone _____

Thanks in advance for helping with our product planning,

Rusty Whitney

Rusty Whitney, Chairman

staple - - -

Editors note: The OPUS memberhip list is not alphabetized. The original format has been retained.

If you wish, you may remove these pages from the newsletter and keep them as a separate packet. To make a 5 by 8 inch booklet, follow these directions:

1. Cut the pages along the inside edge.
2. Staple in two places on the dotted line.
3. Fold around the staples.

To put them in a three-ring binder, you must punch holes along the outer edge and cut them along the inside edge.

Mr. Jerry Shaver
Allergan
2525 Dupont Drive
Irvine,
CA 92713
UW

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RSX-11M
APPLICATIONS: Lab. Interfaces
SYSTEMS: Business

Mr. Dave Wilborn
AmicuSoft, Inc.
113 E. Savarona Way
Carson,
CA 90746
UW (213) 538-4682

COMPILERS: ALL
MACHINES: PDP 11/XX
OPERATING SYS.: V4.0
APPLICATIONS: Business
SYSTEMS: EOM, Business

Mr. Bob Caldwell
Centaurus Software Inc.
975 Hornblend, Suite B
San Diego,
CA 92109
UW (619) 270-4552

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX 11/23, 11/44, 11/24
OPERATING SYS.: RSX-11M
APPLICATIONS: Process Control, Env. Control
SYSTEMS: OEM & Custom Process Control

Mr. Gil Larson
TMI
17862 Fitch
Irvine,
CA 92714
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Martin Greenberger
The Color Press
4871 West Washington Blvd.
Los Angeles,
CA 90016
UW (213) 937-9242

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Suvat Sukchindasathien
Autologic, Inc.
1050 Rancho Conejo Blvd.
Newbury Park,
CA 91320
UW (805) 498-9611 x188

COMPILERS: 2.0
MACHINES: PDP 11/XX (70)
OPERATING SYS.: RSX-11M Plus Version 2.0
APPLICATIONS: Scientific: Compiler, Linker, Library Programs Other: Fonts Programs
SYSTEMS:

Mr. Ralph Haas
1044 Calle Pecos
Thousand Oaks,
CA 91360
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Ms. Christina Runtagh
Telesis
Corporation of Delaware, Inc.
21 Alpha Road
Chelmsford,
MA 01824
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Jeff Hemmerling
Test Systems Strategies
7929 SW Cirrus Dr.
Bearerton
OR 97005
UE (503) 643-9281

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX 23
OPERATING SYS.: RSX-11M
APPLICATIONS: Real Time Process Control
SYSTEMS:

Mr. Donald Andruska
Senior Programmer/Analyst
Honeywell Inc.
Commercial Construction Div.
1500 West Dundee Road
MS 5600
Arlington Heights,
IL 60004
UM (312) 394-4000

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Dick Pearson
5910 Flower
Arvada,
CO 80004
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Pablo Campos
8004 Willis Avenue
Panorama City,
CA 91402
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Thomas Connell
Data Processing Systems
Manager
Koogle & Pouls Eng., Inc.
833RA Comanche, N.E.
Albuquerque,
NM 87110
UW (505) 294-5051

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, 11/34A 11/60
OPERATING SYS.: RSX-11M, RSTS/E
APPLICATIONS: Eng. (mapping), (graphics)
SYSTEMS: Graphics

Mr. Dave Rivard
TestMaster
3191 "D" Airport Loop Dr.
Costa Mesa,
CA 92626
UW (714) 660-0436

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, 11/34, 11/23+ (future)
OPERATING SYS.: RSX-11M 3.2, RSX-11M/Plus (future)
APPLICATIONS: ATE
SYSTEMS: ATE

Mr. Randy Biallas
Medtronic Inc.
3055 Old Hwy. 8
Minneapolis
MN 55440
UM (612) 574-3623

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, VAX, MC6800
OPERATING SYS.: RSX-11M
APPLICATIONS: Medical, Real-time, NC Develop.
SYSTEMS:

Mr. Don Sylwester
System Manager
Concordia College
800 North Columbia Ave.
Seward,
NB 68434
UM (402) 643-3651

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX 11/70
OPERATING SYS.: RSTS/E
APPLICATIONS: Academic
SYSTEMS: Academic

Mr. Mark Berry
Lawrence Livermore Labs
M.S. L330
P.O. Box 808
Livermore,
CA 94550
UW (415) 423-3274

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Pablo Campos
Litton Data System
8000 Woodley
Mail Drop 48-10
Van Nuys,
CA 91409
UW (213) 902-4807

COMPILERS: 1.3, 2.1
MACHINES: PDP 11/XX PDP 11/23 (LSI)
OPERATING SYS.: RT-11 04.00, RSTS/E
APPLICATIONS: Business
SYSTEMS: OEM, Busines

Mr. Ed Costello
Manager, Systems/Software
Helgeson Nuclear Services
5587 Sunol Blvd.
Pleasanton,
CA 94566
UW (415) 846-3453

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Michael Fay
Microtec
P.O. Box 60337
Sunnyvale,
CA 94088
UW (408) 733-2919

COMPILERS: 2.0/2.1
MACHINES: PDP 11/60, VAX 11/750
OPERATING SYS.: RT-11, RSX-11M
APPLICATIONS: Cross-Pascal, C for Micros
SYSTEMS:

Mr. Earl Girbovan
Proquip Inc.
1725 De La Cruz Blvd.
Building 3
Santa Clara,
CA 95050
UW (408) 496-0322

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Rick Freedman
Ross Systems, Inc.
1860 Embarcadero Rd.
Palo Alto,
CA 94061
UW (415) 856-1100

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, VAX
OPERATING SYS.: RSTS/E
APPLICATIONS: Financial Modelling, Graphics, Data Base Mgt.
SYSTEMS:

Mr. Martin Hall
Beenhams Railway Lane
Littlemore
Oxford

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Dave Charlesworth
Systems Analyst
Department of Electronic Services
Campus Services Division
Queen's University
Kingston,
Ontario K7L 3N6
CA (613) 547-6640

COMPILERS: 1.2, 2.0
MACHINES: PDP 11/XX, 11/23
OPERATING SYS.: RT-11 4.0
APPLICATIONS: Office, Process Control, Communications
SYSTEMS:

Mr. Gerry Pelletier
Systems Engineer
Prior Data Sciences Ltd.
39 Highway 7
Ottawa,
Ontario K2H 8R2
CA (613) 820-7235

COMPILERS: 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX, VAX, MC68000
OPERATING SYS.: RT-11, RSX-11M, RSX-11M/PLUS, RSTS/E
APPLICATIONS: Scientific, Academic, Military, Real-time Sys.
SYSTEMS: OEM, Academic, Real-time Sys.

Mr. J.D. Richardson
Defence and Civil Institute
Of Environmental Medicine
1133 Sheppard Ave. W.
P.O. Box 2000
Downsview,
Ontario
CA (416) 635-2073

COMPILERS: 1.1
MACHINES: PDP 11/XX /04,/34,/40,LSI 11/23, VAX (future)
OPERATING SYS.: RT-11 V4 and RT-11 emulator running under UNIX
APPLICATIONS: Scientific, Military
SYSTEMS: Experiment Control & Monitoring

Mr. Christopher Williams
National Research Council
Canada
Low Speed Aerodynamics
Laboratory
Bldg. M-2,Room 125,
Montreal Road
Ottawa,
Ontario K1A 0R6
CA (613) 993-1141

COMPILERS: V2.1A
MACHINES: PDP-11/23, 11/34, 11/44
OPERATING SYS.: RT-11, RSX-11M
APPLICATIONS: Engineering & Scientific, real-time data acquisition & control systems for windtunnels
SYSTEMS:

Mr. Alban Cornish
Senior Systems Analyst
Canada Systems Group
1736 Courtwood Cres.
Ottawa,
Ontario K2C 2B5
CA (613) 225-1171

COMPILERS: 2.1
MACHINES: PDP 11/XX-11/23, VAX-11/780
OPERATING SYS.: RSX-11M 3.2
APPLICATIONS: Scientific: Interactive Centographic Sys., Other: Computer Vehicle Dispatch Sys.
SYSTEMS: OEM

Mr. Oddgeir Uglum
Kokums Camcar Corp.
P.O. Box 5245
Vancouver,
WA 98668
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Joseph Cook
Larse Corporation
4600 Patrick Henry Dr.
Santa Clara,
CA 95050
UW (408) 988-6600 Ext 312

COMPILERS: 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RSX-11M
APPLICATIONS: Telecommunications
SYSTEMS: OEM

Mr. John Chadwick
Sundstrand Data Control
Overlake Industrial Park
Redmond,
WA 98052
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Timothy Dale
System Manager
IDD Test Engineering D/S 61-244
IDD Tektronix
P.O. Box 1000
Wilsonville,
Oregon 97070
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Crawford Brewer
DCT Systems Inc.
15 Archibald Street
P.O. Box 832
Moncton, N.B. E1C 8N6
Canada
CA (506) 389-1222

COMPILERS: 1.2/1.3
MACHINES: PDP 11/XX 11/34
OPERATING SYS.: RSTS/E
APPLICATIONS: RDM Data Mgmt. Sys.
SYSTEMS: Business

Mr. Matt Richards
Research & Development
Polkaudio
1915 Annapolis Road
Baltimor,
MD 21230
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. A.B. Bailey
Systems Consultant
Centre-file Limited
P.O. Box 177
75 Leman Street
London
England E1 8EX
GB 01-488 3131

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Manuel Guerra
Experiencias Industriales,
S.A.
C/O Joaquin Rodrigo
Aranjues,
Madrid
SP (91) 891 0840

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RSX-11M
APPLICATIONS: Business: Dept. Control
SYSTEMS: OEM: Real-time control, ARQ Systems

Mr. Ed Moran
Horace Mann School
231 West 246th Street
Bronx,
NY 10471
UE (212) 548-4000

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, 11/44
OPERATING SYS.: RSTS/E
APPLICATIONS: Academic
SYSTEMS: Academic

Mr. Henry Baird
RCA Lab
Rm 201
Princeton,
NJ 08540
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Mike Dearing
University of Wisconsin
Inst. Sys. Center
1500 Johnson Dr.
Madison,
WI 53706
MW (608) 263-1564

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, VAX
OPERATING SYS.: RT-11, RSX-11M
APPLICATIONS: Scientific, Engrg.
SYSTEMS: OEM, Academic, Research

Mr. Richard Chlopan
Pros. Dir. Comp. Sci.
Westmar College
Le Mars,
IA 51031
MW (712) 546-7081 ext. 315

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, 11/44
OPERATING SYS.: RSTS/E
APPLICATIONS: Business: Admin. Programs, Academic: Comp. Sci. Proj.
SYSTEMS: Academic: Monitors and Tools, etc.

Mr. B.J. Th. Ockeloen
System Engineer
Universiteit van Amsterdam
Vakgroep Algemene Dierkunde
(Dept. of Zoology)
Biologisch Centrum Gebouw II
Kruislaan 320
1098 SM Amsterdam,
Netherlands
NE 020) 680551 Ext 133

COMPILERS: 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX, PDP 11/60, PDP 11/34, MC68000 (soon)
OPERATING SYS.: RSX-11M V.3.2 Autopatch E
APPLICATIONS: Scientific: Zoological, Acedemic: Pattern recognition
SYSTEMS: Academic

Mr. Sydney Davis
Computrol
15 Ethan Allen Highway
Ridgefield,
CONN 06877-6297
UE (203) 544-9371

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Dr. A. Das
Associate Professor
Computer Science
Quincy College
Quincy,
IL 62301
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Prof. Allan Smith
Associate Professor
Drexel University
Department of Chemistry
Philadelphia,
PA 19104
UE (215) 895-2667

COMPILERS: 1.2
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11
APPLICATIONS: Scientific: Chem. Res. Academic: Instructional Software
SYSTEMS:

Mr. Jeffrey Levine
Department of Earth and
Planetary Sciences
The Johns Hopkins University
Baltimore,
MD 21218
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Robert A. Rennert
Director of Academic Computing
Kenyon College
Gambier,
OH 43022
UE (614) 427-2244 Ext. 2559

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Murray Zetterholm
Ford Motor Credit Co.
The American Rd., Room 2235
Dearborne,
MI 48121
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Bob Friedman
E.I. Dupont Company
Clinical Systems Division
Glascow Site Rt 896
Glascow,
DE 19702
UE (302) 453-3280

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, PDP 11/70, 11/44, 11/24, MC68000
OPERATING SYS.: RSX-11M, RSX-11M/PLUS
APPLICATIONS: General Sys. use Downloaders, Utilities, General Math
SYSTEMS:

Mr. Robert Lacovara
InterNet
500 Grand Avenue
Englewood,
NJ 07631
UE (201) 567-3363

COMPILERS: 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX /23
OPERATING SYS.: RT-11,RSX-11M
APPLICATIONS: Mfg. Data Keeping
SYSTEMS: OEM, Bus.

Mr. Robert Thornton
New York Institute of
Technology
Computer Graphics Laboratory
P.O. Box 170
Old Westbury,
NY 11568
UE (516) 686-7644

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Jim Nash
JEOL USA Inc.
11 Dearborne Rd.
Peabody,
MA 01960
UE (617) 535-5900

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Alan Duberstein
Pine Instrument Co.
3345 Industrial Blvd.
Bethel Park,
PA 15102
UE (412) 831-8870

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RSX-11M, RSX-11M/PLUS Versions
APPLICATIONS:
SYSTEMS:

Mr. Hal Laurent
Psych Systems
600 Reisterstown Rd.
Baltimore,
MD 21208
UE (301) 486-2206

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Bob Schor
The Rockefeller Unviersity
1230 York Avenue
New York,
NY 10021-6399
UE

COMPILERS: 1.1, 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11 (4 & 5), TSX-11 (2 & 3)
APPLICATIONS: Scientific: Data Proc., Academic: Algorithm Testing, Teaching
SYSTEMS:

Mr. Steven Bentley
Academic Computing Center
Tougaloo College
Tougaloo,
MS 39174
UM

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Ken Tibesar
3M
3M Center, Bldg.18-1
Saint Paul,
MN 55144
UM (612) 733-8819

COMPILERS: 1.1, 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX, VAX
OPERATING SYS.: RSX-11M, RSX-11M/PLUS, VAX-VMS
APPLICATIONS: Mfg.
SYSTEMS: Process Control

Mr. Troy Monaghen
William Rainey Harper College
Roselle and Algonquin Roads
Palatine,
IL 60067
UE (312) 397-3000

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Terry Medlin
Vice President
GEJAC Incorporated
P.O. Box 188
Riverdale,
MD 20737
UE (301) 8664-3700

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, VAX
OPERATING SYS.: RSX-11M, VMS
APPLICATIONS: Business
SYSTEMS:

Dr. Richard J. Perry
Villanova University
Dept. of Electrical
Engineering
Villanova,
PA 19085
UE (215) 645-4969

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11 4.0
APPLICATIONS: Academic: Communications, Control & Signal Proc.
SYSTEMS: Academic

Mr. Harald Norvik
IKU
Organic Geochemistry Dept.
Hakon Maynussons gt. 1B
Postboks 1883
Norway
7001
NO (7) 915660

COMPILERS: 1.2/1.3
MACHINES: PDP 11/XXPDP 11/23, PDP-11/24
OPERATING SYS.: RT-11 V.4 (slightly), RSX-11M V.3.2 & 4.0 (mainly)
APPLICATIONS: Scientific: chemistry
SYSTEMS: Business

Mr. Mike McCready
Software Manager
Automation Engineering
Div. of Refrig. Eng. Co. Ltd.
26 Great South Road, Otahahu
P.O. Box 12072
Auckland,
New Zealand
NZ (09) 276-8524

COMPILERS: 1.2/1.3 Pascal-1
MACHINES: PDP 11/XX /23
OPERATING SYS.: RSM-11M
APPLICATIONS: Real-time process control
SYSTEMS: Real time process control

Ms. Tellern Asiado
Institute of Advanced Computer
Technology (I/ACT)
SGV Development Center
105 De la Rosa St.
Legaspi Village,
Makati
Phillipines
PH

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Herje Wikegard
SERN
DATOR KONSULT AB
Box 14070
Goteborg,
Sweden
SW +46-31-830800

COMPILERS: 1.3, 2.0/2.1
MACHINES: PDP 11/XX, VAX, MC68000
OPERATING SYS.: RT-11, RSX-11M
APPLICATIONS: Scientific: Tech. Dev. Proj. as Consultants, Other: Real time sys.
SYSTEMS: In house sys. or the customers own sys.

Mr. W.R. Mitchell
Chief Civil Engineer
THE HYDRO-ELECTRIC COMMISSION
4-16 Elizabeth Street
Hobart,
Tasmania
TA 30-1101

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Arthur Brown
10709 Weymouth St.
Garrett Park,
MD 20896
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Paul Bauer
Advanced Control Systems
13809 Industrial Park Blvd.
Minneapolis,
MN 55441
UE

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Bob Sanford
Digital Equipment
New Jersey Turnpike Authority
c/o New Jersey Turnpike
Authority
Administration Bldg.
P.O. Box 1121
New Brunswick,
NJ 08903
UE (201) 247-0900 Ext 233

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX 11/60, 11/40 & 11/04
OPERATING SYS.: RSX-11M
APPLICATIONS: Automatic Traffic Surveillance & Control
SYSTEMS: Real time

Mr. Gary Ware
EECO Incorporated
1601 East Chestnut Ave.
Santa Ana,
CA 92701
UW (714) 835-6000

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Chuck Campbell
Diagnostic Engineering
Digital Equipment Corp.
301 Rockrimmon Blvd. South
Colorado Springs,
CO 80963
UW

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Rev. James Keene
Dir. Computer Center
Saint Louis University High School
Backer Memorial
4970 Oakland Avenue
St.Louis,
MO 63110
UW

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX PDP 11/23
OPERATING SYS.: RT-11, & TSX-PLUS
APPLICATIONS: Academic: Teaching & In School Appl.
SYSTEMS: Academic

Mr. Phillip Ricker
Engineering Test Manager
Intel Corporation
Components Division
3585 S.W. 198th
M/S F4-2-546
Aloha,
OR 97007
UW (503) 642-6592

COMPILERS: 1.2/1.3
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11
APPLICATIONS: Semi-conclution Testing, Cont. Sys. Proc.
SYSTEMS: In-House

Mr. John Hallick
Manager Partner
Miller Lucas Company
Systems & Programming
4811 N. Sterling Ave.
Peoria
IL 61615
UM (309) 692-5640

COMPILERS:
MACHINES:
OPERATING SYS.:
APPLICATIONS:
SYSTEMS:

Mr. Pat D'Andrea
Sr. Software Engineer
MCC Powers
2942 MacArthur Blvd.
Northbrook,
IL 60062
UW (312) 272-9555

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX, MC68000
OPERATING SYS.: RSX-11M
APPLICATIONS: Energy Mgmt/ Temperature Controls
SYSTEMS:

Mr. Steven Brecher
Software Supply
4618 E. 6th Street
Long Beach,
CA 90814
UW (213) 434-3723

COMPILERS: 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11 4.0, RSX-11M 4.0
APPLICATIONS: Support of Oregon Software Customers
SYSTEMS: OEM

Mr. Ronald Webster
Computer Services
Arizona State University
ECA 108
Tempe,
AZ 85287
UW (602) 965-1203

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RSTS/E
APPLICATIONS: Acedemic
SYSTEMS: Acedemic

Mr. Glenn Stearns
Software Development Manager
Microvertics Corp.
1394 Shorebird Way
Mountain View,
CA 94043
UW (415) 961-9454

COMPILERS: 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11 4.00B, RSX-11 4.0
APPLICATIONS: Bus. Mgmt. Sys.
SYSTEMS: Sell to end user

Mr. Peter Schmitz
Software Engineer
GenRad S.P.D.
4620 N. 16th Street
Phoenix,
AZ 85016
UW (602) 264-2475

COMPILERS: 1.2
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11
APPLICATIONS: Scientific
SYSTEMS: OEM

Mr. Martin W. Sivula
System's Manager
Lunenburg Public Schools
1079 Massachusetts Ave.
Lunenburg,
MA 01462
UE (617) 582-9941 Ext 4

COMPILERS: 1.2/1.3
MACHINES: PDP 11/XX
OPERATING SYS.: RT-11, RSTS/E
APPLICATIONS: Scientific: Some, Business, Academic
SYSTEMS: Academic

Mr. Abe Getzler
Systems Analyst
Brooklyn Union Gas
195 Montague St.
Brooklyn,
NY 11201
UE (212) 403-2428

COMPILERS: 1.1
MACHINES: PDP 11/XX
OPERATING SYS.: RSX-11M, IAS
APPLICATIONS: Mobile Dispatching
SYSTEMS: Motorola

Mr. John Norman
Academic Computing
Grinnell College
Grinnell,
IA 50112
UM (515) 236-2570

COMPILERS: 1.2/1.3, 2.0/2.1
MACHINES: PDP 11/XX 11/70
OPERATING SYS.: RSTS/E
APPLICATIONS: Coursework, Computer assisted instruction
SYSTEMS: Academic

Mr. Jerry Pitzl
Associate Professor
MACALESTER COLLEGE
1600 Grand Ave.
Saint Paul,
MN 55105
UM (612) 696-6291

COMPILERS: 2.0/2.1
MACHINES: PDP 11/70, VAX (perhaps '83)
OPERATING SYS.: RSTS/E
APPLICATIONS: Academic
SYSTEMS: Academic

Mr. Alex Neussendorfer
Control Data Corp.
3965 Meadowbrook Rd.
St. Louis Park,
MN 55426
UM (612) 935-0366

COMPILERS: 2.1
MACHINES: PDP 11/XX, PDP 11-34, PDP 11-44
OPERATING SYS.: RSX-11M 4.0, 4.1
APPLICATIONS: Business: (Printed Circuit Board Mfg.)
SYSTEMS: Buxiness

Mr. Scott Snadow
General Dynamics
P.O. Box 2507
Mail Zone 4-68
Pomona,
CA 91769
UM (714) 620-7511 Ext 4779

COMPILERS: 2.0/2.1
MACHINES: PDP 11/XX
OPERATING SYS.: RSX-11M
APPLICATIONS: Scientific: Various, Other: Sys.-Related
SYSTEMS:

Ms. Cynthia Dunn
Applied Automation, Inc.
Pawhuska Road 205 ARB
Bartlesville,
OK 74004
UM (918) 661-1915

COMPILERS: 2.0
MACHINES: MC68000-EXORmacs
OPERATING SYS.: VERSAdos
APPLICATIONS: Scientific
SYSTEMS: EOM

Pascal
NEWSLETTER
OREGON SOFTWARE

6915 SW Macadam Avenue
Portland, Oregon 97219

Pascal
NEWSLETTER

NUMBER 10 OREGON SOFTWARE SPRING/SUMMER 1985

ULTRIX compiler joins the Pascal—2 family

The last six months have been an intense period of product development at Oregon Software.

Recent developments include the completion of Pascal—2 for VAX/ULTRIX-32; completion of the Oregon Linker/Assembler for our cross-development systems to the 68000; and completion of native SourceTools for VMS.

Once Pascal—2 for VMS was completed in the fall, we were able to quickly produce the ULTRIX compiler. Our technical team, led by Bruce Graham, used the compiler's common front end with the VAX/VMS code generator to produce a compiler that was retargeted for the ULTRIX system. With this cross-compiler, Bruce then compiled the M68000 UNIX support library, which is itself written in Pascal. This produced a Pascal support library for ULTRIX, which provided a testing environment for larger, more complicated programs. The ULTRIX product was put into field test in April and is being released to end users now.

Completion of the Oregon Linker/Assembler frees us from reliance upon the Oasys cross-linker/assembler,

VAX/VMS compiler retargeted for ULTRIX

which because of its slowness and incompatibility with standard M68000 object format was a weak link in our cross-tools packages from VMS and RSX. The Oregon Linker/Assembler runs under RSX and VMS native mode, making the cross-tools package available on all RSX and VMS configurations. In addition, our Linker/Assembler is designed to be largely machine-independent so that we can move it to new configurations without having to rely on other vendors' software in the future. In addition, the new versions of the cross-compiler itself (2.0P, shortly to be followed by 2.0Q) contain a number of bug fixes.

Native VMS SourceTools also completed field-testing and is being moved into regular distribution. Customers who purchased the RSX-to-VMS upgrade should be receiving their VMS versions shortly, if they haven't already.

We also completed the last round of bug fixes on Pascal—1, which is now an unsupported product. (See "Pascal—1 goes unsupported" later in this newsletter for update details.)

Long-term development

A major goal of Oregon Software has been to make our products — as well as our customers' — portable. Because most of our products are written in Pascal, we have largely succeeded. Our efforts to quickly move Pascal—2 to a variety of different machine architectures, however, have sometimes resulted in compromise. Over time, compiler sources have drifted apart as unique machine-oriented features and bug fixes have been added. For example, in order to make our compilers interface in a natural way with various operating systems, we have had to modify or rewrite command-line processing code for each machine. This has gradually increased maintenance and development problems.

Over the last few months, our compiler developers have been working on the "Common Front End" project. Their goal is to realign the source code for the compiler to provide a clear distinction between machine-dependent and machine-independent code.

The "front end" of the compiler is the code that scans Pascal source programs, analyzes syntax, and does the machine-independent optimizations. Th "back end" of the compiler is the code generator, which is unique to each machine. The code generators for our different products are now being interfa ed with the new common front end.

The common front end code served as the basis for the VAX/ULTRIX and the new 80186/286 project. Now that ou compilers have converged on a common set of sources, maintenance and develop ment will be much easier. Any new feature or bug fix in the front end code will immediately become available in each of the other compilers.

We began the 80186/286 project in April and now have a Pascal—2 compil

'Common front end' improves maintenance and development

for the IBM PC/AT running XENIX. V have a large memory model for code siz that takes advantage of the expanded instruction set of the 80286 processor and we are currently working on the co generation for 32-bit integer arithmetic. Our goal is to have an AT-based compil

Continued on Page

In this issue . . .

President's message . . . . . Page 2
Pascal—1 goes unsupported . . . . . Page 3
Using virtual memory with Pascal programs . . . . . Page 4
Oregon Software documentation . . . . . Page 8
Book Review . . . . . Page 9
Calling VMS system and run-time library routines . . . . . Page 10
Errors, additions to the manuals . . . . . Page 12
The Log . . . . . Page 15
Newsletter Index . . . . . Page 21

Common front end fosters modularity

Oregon Software's technical staff has been hard at work the past few months. Perhaps most exciting has been the work done to consolidate the various Pascal—2 compilers into one uniform structure, isolating machine-dependent code and sharing source modules to a much larger extent than before.

This work is significant for two reasons. First, expanding the use of common source modules will speed bug fixing, increasing the reliability of our Pascal—2 products. A good example is our recent effort to make Pascal—2 comply 100 percent with the ISO standard (past versions have been very close, but not perfect). As a result of the shared source code, most of our Pascal—2 products can be made to comply at a fraction of the effort necessary in the past.

This work will also lessen the effort required to introduce new compiler products. For example, Oregon Software recently announced availability of Pascal—2 for VAX/ULTRIX. The VAX/VMS compiler and VAX/ULTRIX compiler differ in only two areas: the user interface (`program/debug` vs. `program -debug`), and the object/assembly code output routines for the code generator. We have truly modularized the compiler and have made heavy use of MAKE (a component of SourceTools) to manage the system. For instance, the VAX/ULTRIX project began with the implementation of a VAX/VMS to VAX/ULTRIX cross-compiler. After this compiler was completed, it only required slight modification of the MAKE file to replace VMS-host specific modules with ULTRIX-host specific modules to generate a VAX/ULTRIX-hosted native compiler.

Likewise, our Intel 80186/286 product (which has just entered field test) currently runs in native mode under PC-IX, as a cross-compiler under VAX/VMS, and as a cross-compiler under VAX/ULTRIX. These three versions are built by simply including the right pieces for each environment.

The net result of this effort? Better maintainability, as well as a quicker pace in development of new host-target environments supported by Pascal—2, which together mean better service to our customers.

Don R. Baccus, President

New editor named

We're changing editors again! Collins Hemingway began the *Oregon Software Pascal Newsletter*. In the fall of 1983, I took over from him (Number 7). With this issue, Tom Hanrahan becomes our newsletter's editor.

Tom has worked on our user manuals since February 1984 and, until recently, he was managing editor of the *Portland Family Calendar*, a very popular monthly newspaper. Tom has produced this issue in a new format and included an index to previous issues. He has some good ideas for forthcoming issues, too. But, one thing never changes: he'll need articles about your applications, your technical tricks and discoveries, your Pascal-related interests. We can't have a users' newsletter without submissions from users.

New employees draw on wide experience

Software engineer **Jan de Rie** joined Oregon Software in December. He comes to us from the Netherlands, where he worked for almost four years in Leiden. There, he co-introduced Pascal—2 in a large organization (more than 100 programmers) and implemented the Pascal—2 support library on the company's proprietary operating system. Jan attended Erasmus University in Rotterdam, where he earned an M.A. in mathematical economics. Jan's main interest at Oregon Software is working with the Pascal compilers. Currently, he is developing a version of the Pascal—2 compiler for the Intel 80186/286 family of processors. Jan actively pursues his hobby of folkdancing and performs with a dance group around the northwest area.

As our newest account executive, **Ruby Haughton** will soon be introducing herself to many of our customers. Ruby came to Oregon Software in May from the sales department at Proactive, a division of Pacific Telecom Co. A graduate needed to provide new prospects with technical product descriptions and to respond to our current customers' requests for help and additional product information. Outside of Oregon Software, Ruby works with adolescents as a volunteer counselor, and she enjoys singing. She'll be touring the Caribbean this summer as a counselor/performer of an international sports evangelical team.

Steve Bihler joined Oregon Software in mid-January as Technical Support Manager. His primary function will be assisting customers with any technical problems they may encounter. Steve's previous job was as a Senior Systems Engineer at Perkin-Elmer's Technical Support Center. He has an interdisciplinary degree from Washington State University in computer science and business administration. He likes flying, skiing, and radio communications. Steve has lived in more than 24 different places around the world including Greece and the Azores, and he

OPUS Communiqué

Oregon Pascal Users Society

Bruce Williams, who served as the OPUS coordinator since the group's formation in November 1982, has given up his position as head of the Society. Any OPUS member interested in taking over as the Society's coordinator should contact:

Tim McMenamin
Oregon Software
6915 SW Macadam Ave.
Portland, OR 97219
(503)245-2202

The principal duties of the OPUS coordinator are to:

- Collect information submitted by OPUS members and disseminate it through this column.
- Facilitate contact among OPUS members by matching one member's problems with another member's expertise.
- Maintain and coordinate the OPUS shared library.

Over the past few years, OPUS has accomplished many of its initial goals. Membership has grown to over 100 individuals from around the world. The Society has helped numerous users of Pascal—1 and Pascal—2 with programming problems—everything from simple character I/O to asynchronous-trap routines, and most recently, the Society has taken steps to establish a shared library of routines and utility programs.

But clearly, the continued success of the Oregon Pascal User Society depends on the efforts of its membership and its ability to draw upon its membership to fill key positions within the organization.

Pascal—1 goes unsupported

Pascal—1 version 1.3D was released for field test on April 8, 1985. Version 1.3D is the final release of our Pascal—1 compiler. The software is stable in its present form, and Oregon Software plans no subsequent releases.

Pascal—1 is Oregon Software's first Pascal compiler. Written entirely in MACRO-11, the compiler runs on DEC's PDP-11 series machines. It implements Pascal in much the same form as originally conceived by Jensen and Wirth with some extensions, such as separate compilation facilities.

Pascal—1 has been in commercial use since 1977. Thanks to its quick compilation and its minimal memory demands, Pascal—1 was rapidly embraced by the academic community, where it still enjoys wide use.

In the interest of portability, improved code optimization, and adherence to the international Pascal standard (ISO 7185), Oregon Softwa troduced its new compiler, Pascal—2 summer 1981. As Pascal—2 far excee Pascal—1 in performance and portat we propose to devote our resources t maintaining and enhancing the newe compiler.

Oregon Software will no longer support contracts for Pascal—1. Customers who were under support Pascal—1 as of January 1984 are elig to receive the final update free of ch and should contact our customer ser department to request their upgrade. Customers who are not eligible for t free update and wish to purchase the software should call our sales staff f price information and product availability.

Pascal
NEWSLETTER
OREGON SOFTWARE
6915 SW Macadam Avenue
Portland, Oregon 97219

Thomas E. Hanrahan, editor
David Spencer, David Barnes, Collins Hemingway, writers
Betsy Slonaker, production assistant

The Pascal Newsletter is published regularly by Oregon Software, Inc., 6915 SW Macadam Ave., Portland, OR 97219; (503) 245-2202. Each customer of Oregon Software receives one free subscription per site. Additional subscriptions are available upon written request.

The Pascal Newsletter accepts articles of interest to Pascal users: solutions to troublesome programming situations, new applications of Pascal, interesting variations on standard applications, etc. Submit articles on paper (typed and double-spaced), on floppy disk, or on magnetic tape, sent to the attention of th editor. We pay $100 per newsletter page for any article we print.

RSX virtual memory window speeds record access

by Steve Poulsen

RSX system services known as memory management directives control the way in which memory management hardware of a PDP-11 computer maps a program's virtual memory address space to physical memory. In general, Pascal programs running on a mapped RSX system have access to a maximum of 64K bytes of memory. The limitation is imposed by the 16-bit nature of PDP-11 computers — at any instant, a program may not have direct access to more than 64K bytes of virtual address space. But physical memory on most PDP-11 computers is considerably larger than 64K bytes and as long as you do not attempt to directly reference more than 64K bytes at a time, you can write Pascal programs that call upon the memory management directives to take advantage of additional memory resources.

The physical memory accessible to a task is called a region. (Shared libraries and shared common areas are examples of regions.) The block of physical memory in which a task runs is called the "task region." Access to regions is through "windows," which are a portion of a task's virtual memory. Each window must start at an address that is a multiple of 4K words. So, in any 32K word task,

Programs can access additional memory on most PDP-11s

up to 8 windows may be defined. The size of a window may vary from 32 words to 32K words. If a region is larger than the size of a window, the window may be moved to different locations within the region. For example, a 4K word window may be mapped to many different physical addresses within a 100K word region. The physical size of a region is limited only by the size of the partition in which the region resides.

To use virtual memory, a Pascal program must call upon the RSX memory management directives to:

- Attach to an exiting or dynamically
- Create an address window to use in accessing a region.
- Map a window into a region.

The RSX memory management directives (described in the *RSX-11M/PLUS Executive Reference Manual*) may be called directly by a Pascal progam through FORTRAN entry points in your system library (LB:[1,1]SYSLIB.OLB). To provide an interface between a Pascal program and the RSX directives, you must create two special data structures: a region definition block and a window definition block, both of which may be represented as simple Pascal records. The fields for these two structures correspond to the block parameters defined for specific memory management directives described in the executive reference manual.

The sample program VIRT.PAS, presented on the following page, is an example of how to call RSX memory management directives from a Pascal program. Explanations within the text of the code appear as ordinary Pascal comments. As the example shows, the program contains Pascal record definitions for the region definition and window definition blocks used by the system services. In addition, VIRT.PAS contains the nonpascal procedure declarations required in calls to the memory management directives.

The subroutines in VIRT.PAS implement a simple virtual file system by creating a dynamic region and then reading and writing data records within the region in much the same way that a disk file operates. The VINIT procedure initializes the virtual memory system by creating a dynamic region called PASDAT in the GEN partition. The user specifies the size of the region. VINIT also creates an address window that maps APR7 into the dynamic region. This causes virtual addresses in the range 160000 to 177777 to be mapped into the dynamic region.

To use VIRT.PAS, you must first modify the code to define the type of data to be stored in the virtual file. An integer is used for demonstration purposes in the current example. Access to the records in virtual memory is through VPUT. These procedures each use a record number to compute the physical location of the desired record in the dynamic region. Then, if necessary, a window is mapped into the region so that the desired record is contained within the window. This makes the record available to the task. The data can be read, updated in place, or copied to some other record in the task. In its current form, the VIRT program must be compiled

Virtual file systems are very efficient

separately from its calling program because it uses OWN storage to maintain the current mapping information. The routine can easily be adapted for use as an include module by removing the $own directive.

The Task Builder cannot predict how many windows your program might create. So, when you task-build a program that uses virtual memory, you must instruct the Task Builder to build additional window data structures. The WNDWS = *n* option creates additional window blocks in the task's header. For example, if your program is called TEST, you could task-build it with the following commands:

```
> TKB
TKB > TEST/FP/CP = TEST,VIRT,
TKB > LB:[1,1]PASLIB/LB
TKB >/
ENTER OPTIONS:
TK > WNDWS = 1
TKB >//
```

When you create windows, you should base them at high virtual addresses. In the current example, APR 7 (the highest available APR) is used for the mapping. If your task maps to a resident library such as PASRES, you may need to use a tower APR for the window because Pascal tasks extend the size of the task (window 0) when additional heap space is required. When you are also using virtual memory for data or for virtual overlays, it is possible that the program may accidentally extend into virtual address already being used for

```
PROGRAM virt;

CONST
    apr = 7; { APR to use for mapping }
    blockettes_per_4kw = 128; { 8192 div 64 }

TYPE
    data = integer; { Simple test case }
    data_pointer = ^data; { Pointer to data record }
    byte = 0..255; { Unsigned byte }
    word = 0..65535; { Unsigned word }
    name = PACKED ARRAY [1..6] OF char; { Name to convert to rad50 }
    rad50_name = ARRAY [1..2] OF word; { 6 rad50 chars }
    directive_status = word; { Directive status returned from exec call }
    region_id = word; { System supplied region ID }
    region_status_bits = (rs$red, rs$wrt, rs$ext, rs$del, rs$nex, rs$att,
                          rs$ndl, rs$mdl, rs$xx1, rs$xx2, rs$xx3, rs$xx4,
                          rs$xx5, rs$xx6, rs$unm, rs$crr);
    region_status = PACKED SET OF region_status_bits;
    access_bits = (pread, pwrite, extend, delete);
    access_category = (system, owner, group, world);
    accesses = PACKED SET OF access_bits;
    protection_word = PACKED ARRAY [access__category] OF accesses;

    region_block =
      RECORD
        gid: region_id; { System supplied region id }
        gsiz: word; { Size of region in blockettes [64 bytes each] }
        gnam: rad50_name; { Region name [or zeroes] }
        gpar: rad50_name; { Partition in which region resides }
        gsts: region_status; { Region control & status bits }
        gpro: protection_word; { Region protection word }
    END;

    window_status_bits = (ws$red, ws$wrt, ws$ext, ws$del, ws$bps, ws$sis,
                          ws$rcx, ws$map, ws$64b, ws$nat, ws$res, ws$nbp,
                          ws$rrf, ws$elw, ws$unm, ws$crw);
    window_status = PACKED SET OF window_status_bits;

    window_block =
   PACKED RECORD
     nid: byte; { System supplied window id }
     napr: byte; { APR to use to map virtual addresses }
     nbas: word; { Virtual base address of window [APR * 8192] }
     nsiz: word; { Size of window }
     nrid: region_id; { ID of region which this window maps into }
     noff: word; { Offset [in blockettes] into region }
     nlen: word; { Length [in blockettes] to map [0 = default] }
     nsts: window_status; { Window control & status bits }
     nsrb: word; { Send/receive buffer address [not used] }
END;

VAR
    data_size: word; { Size in bytes of DATA }
    max_record: word; { Max record number }
    records_per_4kw: word; { Number of data records in each 4KW block }
    current_4kw_block: word; { Block currently mapped }
    rdb: region_block; { Region definition block }
    wdb: window_block; { Window definition block }

PROCEDURE atrg(VAR rdb: region_block;
               VAR status: directive_status);
  NONPASCAL; { Attach region }

PROCEDURE crrg(VAR rdb: region_block;
               VAR status: directive_status);
  NONPASCAL; { Create region }

PROCEDURE dtrg(VAR rdb: region_block;
               VAR status: directive_status);
  NONPASCAL; { Detach region }

PROCEDURE craw(VAR wdb: window_block;
               VAR status: directive_status);
  NONPASCAL; { Create address window }

PROCEDURE elaw(VAR wdb: window_block;
               VAR status: directive_status);
  NONPASCAL; { Eliminate address window }

PROCEDURE map(VAR wdb: window_block;
              VAR status: directive_status);
  NONPASCAL; { Map address window }

PROCEDURE umap(VAR wdb: window_block;
               VAR status: directive_status);
  NONPASCAL; { Unmap address window }

PROCEDURE cvt_rad50(n: name;
                    VAR r: rad50_name);

  FUNCTION rad50(a, b, c: char): word;

    FUNCTION rad(c: char): word;

      BEGIN { Rad }
        IF c = ' ' THEN rad := 0
        ELSE IF c IN ['A'..'Z'] THEN rad := ord(c)—ord('A') + 1
        ELSE IF c IN ['0'..'9'] THEN rad := ord(c)—ord('0') + 30
        ELSE IF c = '$' THEN rad := 27
        ELSE IF c = '.' THEN rad := 28
        ELSE IF c = '_' THEN rad := 29
        ELSE
          BEGIN
          writeln(' *', c, '* is not a legal RAD50 character!');
          rad := 0;
          END;
    END; { Rad }

  BEGIN { Rad50 }
    rad50 := (rad(a) * 40 + rad(b)) * 40 + rad(c);
  END; { Rad50 }

BEGIN { Cvt_Rad50 }
  r[1] := rad50(n[1], n[2], n[3]);
  r[2] := rad50(n[4], n[5], n[6]);
END; { Cvt_Rad50 }
```

The program VIRT.PAS provides an example of how to use virtual memory from Pascal. The subroutines in VIRT.PAS implement a virtual file system that is more efficient and has faster accessibility than a similar file system maintained on disk.

```
PROCEDURE vinit(number_of_records: word);
  EXTERNAL;

PROCEDURE vinit;

  VAR
    n_4kw_blocks: word; { Number of 4KW blocks needed to hold virtual data }
    remainder: word; { Blockettes required in final portion }
    status: directive_status; { Directive status }

  BEGIN { Vinit }
    data_size := size(data);
    IF odd(data_size) THEN data_size := data_size + 1;
    max_record := number_of_records;
    records_per_4kw := 8192 DIV data_size;
    n_4kw_blocks := number_of_records DIV records_per_4kw;
    remainder := ((number_of_records MOD records_per_4kw) * data_size +
                  63) DIV 64;

    WITH rdb DO
      BEGIN
      gid := 0;
      gsiz := n_4kw_blocks * blockettes_per_4kw + remainder;
      cvt_rad50('PASDAT', gnam); { Region name }
      cvt_rad50('GEN ', gpar); { Partition containing region }
      gsts := [rs$att, rs$wrt, rs$red, rs$mdl];
      gpro[system] := [ ];
      gpro[owner] := [ ];
      gpro[group] := [delete, extend, pwrite];
      gpro[world] := [delete, extend, pwrite, pread];
      END;
    crrg(rdb, status);
    IF status < > 1 THEN
      writeln('Can''t create region. Status = ', status: 1, ' . ');
    WITH wdb DO
      BEGIN
      nid := 0;
      napr := apr;
      nbas := 0;
      nsiz := blockettes_per_4kw;
      nrid := rdb.gid;
      noff := 0;
      nlen := 0;
      nsts := [ws$wrt];
      nsrb := 0;
      END;
    craw(wdb, status);
    IF status < > 1 THEN
      writeln('Can''t create address window. Status = ', status: 1, ' . ');
    current_4kw_block := -1;
  END; { Vinit }

FUNCTION vseek(record_number: word): data_pointer;
  EXTERNAL;

FUNCTION vseek;

  VAR
    needed_4kw_block: word; { Block containing desired record }
    offset_in_4kw_block: word; { Position of record in block }
    status: directive_status; { Status of MAP directive }

  BEGIN { Vseek }
    record_number := record_number - 1; { Make relative to zero }
    IF (record_number < 0) OR (record_number > max_record) THEN
      BEGIN
      writeln('Virtual record number ', record_number + 1: 1,
              ' is out of range');
      needed_4kw_block := 0;
      offset_in_4kw_block := 0;
      END
    ELSE
      BEGIN
      needed_4kw_block := record_number DIV records_per_4kw;
      offset_in_4kw_block := (record_number MOD records_per_4kw) *
                              data_size;
      END;
    IF needed_4kw_block < > current_4kw_block THEN
      BEGIN { Map new block containing desired record }
      wdb.noff := needed_4kw_block * blockettes_per_4kw;
      wdb.nlen := 0;
      map(wdb, status);
      IF status < > 1 THEN
        writeln('Can''t map to record ', record_number: 1, '. Status = ',
                status: 1, ' . ')
        ELSE current_4kw_block := needed_4kw_block;
        END;
    vseek := loophole(data_pointer, wdb.nbas + offset_in_4kw_block);
  END; { Vseek }

PROCEDURE vget(record_number: word;
               VAR d: data);

  EXTERNAL;

PROCEDURE vget;

  VAR
    p: data_pointer;

  BEGIN { Vget }
    p := vseek(record_number);
    d := p^;
  END; { Vget }

PROCEDURE vput(record_number: word;
               VAR d: data);

  EXTERNAL;
PROCEDURE vput;

  VAR p: data_pointer;

  BEGIN { Vput }
    p := vseek(record_number);
    p^ := d;
  END; { Vput }
```

You can prevent Pascal tasks from using additional virtual memory by means of a global patch. The global variable P$NEXT can be patched to contain a value of 240 to prevent Pascal from extending into additional virtual addresses, as shown:

```
> TKB
TKB > TEST/FP/CP = TEST,VIRT,
TKB > LB:[1,1]PASLIB/LB
TKB >/
ENTER OPTIONS:
TKB > GBLPAT = TEST:P$NEXT:240
TKB > EXTSCT = $$HEAP:20000
TKB >//
```

In earlier versions of Pascal this symbol is named $NOEXT.

When you prevent Pascal from using additional memory, you must provide explicit dynamic memory for the program by extending the heap via the EXTSCT option as shown in the previous task-build example. For each program, the amount of memory required for heap storage varies according to the amount of local variable storage and the number of files your program opens. If your program dies with a stack overflow or with a not enough memory error, you should increase the size of the heap.

With the patch just described, your Pascal program will never use any additional virtual memory. This gives you complete control over the allocation of virtual memory within the task.

One of the principal advantages gained by using virtual memory is speed. In the case of the virtual file system described here, records are read and written by copying the data from the task's memory into the memory of a dynamic region. Records are accessed by moving a region's mapping window instead of by reading a disk file. This memory-to-memory copy operation is very efficient and can be done in a matter of micro-seconds rather than milli-seconds.

Steve Poulsen is currently manager of Oregon Software's technical sales support. As one of the founders of Oregon Software, Steve has contributed heavily over the years to the company's development effort.

Rites of Spring . . .

Party Thrives in New Setting

Oregon Software's annual corporate party on May 17 moved indoors for the first time this year. Previously, we had always celebrated the company's incorporation at our old building on Canyon Road, where ample grounds and a secluded street made the commemoration a unique and well-attended event within the community. There were skeptics among those attending past celebrations who believed that the flavor of the old days would be lost in our new facility.

Like a die-hard C programmer being explained the benefits of Pascal, the skeptics were certain that such a conversion could never be made. What was going to happen to the bountiful barrels of imported beer that graced the halls of the previous parties? Probably replaced by wine coolers. What about the live folk music of the olden days? Probably done in by piped-in music from the dentist office down the street. Doubters were quickly reassured, however, as the party's featured event, Oregon Software's first-ever "Chili Cook-Off," immediately set the tone for much of the evening's activities.

An entire validation suite of tests was run on those entries submitted. The judges savored each dish and carefully cleansed their palates with Dos XX beer before moving on to the next entry. The dishes were tested for hotness, greasiness, viscosity, taste, and originality. After a long discussion, the judges decided that Bob Phillips, scoring 94 out of a possible 100 points, had won. The competition was extremely close, but Bob's chili had the robustness of a Pascal—2 compiler, and that in the end made the difference.

Competition among the finalists in Oregon Software's first Chili Cook-Off was close, but Bob Phillips's "Rump-po entry was named the contest's winner.

First prize was a night for two at Neskowin Resort on the Oregon coast.

After the judging, party-goers were invited to decide for themselves which c the chilies they preferred, as they drank margaritas and listened to mandolin fol music performed by Sandy Profeta and Jan DeWeese.

RSTS/E Version 9 User's Note

We recently received our copy of the latest RSTS/E release from Digitial Equipment Co. and have begun the process of testing and converting Pascal—2 to run under Version 9 of the operating system. We will announce the release of any new software required to run on Version 9 as soon as it is ready for distribu tion. Customers with current support contracts who are upgrading to Version can request the new release of Pascal— as soon as our announcement of its availability is made.

Publication list grows as product list expands

by David Spencer

When I started at Oregon Software a little over three years ago, we had three technical writers, three existing manuals, and one new manual in progress. Today, we have three writers, seventeen existing manuals, and a new one in progress. Maintaining and improving our manuals demands more of our resources than producing new documentation.

A case in point: About two years ago, we announced that new editions of the Pascal—1 user manuals for the PDP-11 products were forthcoming (*Pascal Newsletter* Number 6, Spring 1983). What started out as a six-month project took a lot longer. While working on the PDP-11 books, we created manuals for Pascal—2 on three new microprocessors, for SourceTools on three DEC operating systems, and for the multi-optioned cross-development system on two hosts. With the publication of the Pascal—1 V1.3 manuals in April, we've fulfilled our two-year-old promise and completed basic documentation on all current products.

Despite staff turnover and size limitations, the Technical Publications group plans to improve and extend those manuals as part of its routine maintenance. New sections on the support library, on floating-point format, and our implementation of Pascal are already in progress. We're experimenting with larger typefaces and more white space in our page layouts. Late in the summer, we will begin another cycle of comprehensive revisions.

As we plan the next round of upgrades, we're looking through our files of users' suggestions and the feedback from our distributors' workshops. We're coordinating our efforts with the newly formed support group to gather further indications of your needs. We're developing better methods of production and graphic presentation, in order to make our manuals easier to read and use. We need your feedback on our manuals to continue improving our product documentation. This is an opportune time to send us that suggestion you've always thought you'd make.

OREGON SOFTWARE USER MANUALS: When we last printed a list of manuals and prices in this newsletter, we listed only Pascal—1, Pascal—2, SourceTools, Sort-1-Plus, and the concurrent package. We are again publishing a complete list of user manuals available. Prices and ordering information are provided in the Oregon Software Catalog of Products and Price Schedule *(February 1985).*

Oregon Software User Manuals

Title	Publication Date
Pascal—2 V2.1 for RSX	July. 83*
Pascal—2 V2.1 for RSTS	Aug. 83*
Pascal—2 V2.1 for RT-11	Aug. 83*
Pascal—2 V2.1 for VMS	Oct. 84*
Pascal—2 V2.1 for VAX/UNIX	May 85
Pascal—2 V2.1 for UNIX/68000	Aug. 83
Pascal—2 V2.1 for RT-11/Pro300	Feb. 85
Pascal—2 V2.1 for POS/Pro300	Feb. 85**
SourceTools V1.0 for VMS, RSX, RSTS	Apr. 85
Pascal—2 V2.0 for VERSAdos	Nov. 82*
Pascal—2 V2.0 Cross-Development RSX, VMS	Mar. 84*
Oregon Linker/Assembler RSX, VMS	May 85
Pascal—2 Concurrent Programming Pkg.	Apr. 84*
Pascal—1 V1.3 for RSX	Mar. 85
Pascal—1 V1.3 for RSTS	Mar. 85
Pascal—1 V1.3 for RT-11	Mar. 85
Sort-1-Plus V1.7	Apr. 80

* Items marked with an asterisk are scheduled for an update within the next three months.

** Abbreviated manual.

Pascal—2 compiler available on ULTRIX

Oregon Software's Pascal—2 compiler now runs on ULTRIX-32, Digital's version of the UNIX operating system for VAX and MicroVAX computers, and on Berkeley UNIX (BSD4.2), from which ULTRIX is derived.

As with all Pascal—2 releases on Digital systems, Pascal—2 on ULTRIX-32 includes a high-level interactive debugger, an execution profiler and other utilities that aid in reading and

tional Pascal standard (ISO 7185) and produces extremely concise, fast object code, as compared to that produced by languages such as FORTRAN-77 and C. Pascal—2 supports all features of standard Pascal, including packed data structures and set types of up to 256 elements. The compiler allows calls to separately compiled routines written in Pascal, C, FORTRAN or MACRO, giving

Pascal standard, locates many uninitialized variables and detects nearly 150 Pascal syntax errors.

ULTRIX-32 is a licensed version of Berkeley UNIX that supports the entire BSD4.2 command set, including virtual memory demand paging. Different versions of ULTRIX-32 run on MicroVAX and VAX-11/700 Series computers.

Pascal—2 for VAX/ULTRIX is now

Book Review

by David M. Barnes

David M. Chess, *Programming in IBM PC DOS Pascal*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1985.

Programming in IBM PC DOS Pascal is written for the Pascal programmer developing applications on the IBM PC. The language described is PC DOS Pascal, versions 1.0 and 2.0, which is sometimes referred to in the text as "Pascal 2," by the way. Don't be fooled !

Because Pascal is an excellent teaching language, many Pascal books take the form of programming tutorials. In *Programming in IBM PC DOS Pascal*, the reader is referred to a standard introductory Pascal text for language details. This allows the author, who seems to be in touch with the needs of the applications programmer, to concentrate on the extensions of this dialect of Pascal and its interactions with the PC DOS operating system.

The book is divided into three parts. Part One is "Essential Pascal." It deals with the language itself, introducing the extensions peculiar to this dialect. The author accurately describes this part of the book as a "whirlwind tour." Part Two is "The Details," about the same length as part one, and deals with data files, screen handling, keyboard I/O and other DOS services, such as the assembler interface. This is the main part of the book. While it may slight some programmers' pet topics, it covers many important things that a developer needs to know. Part Three is built around a small programming project that ties in some of the earlier examples in the book. Mr. Chess also touches on some of the mandatory Pascal subjects: style, modularity, and top-down design. On these subjects he is refreshingly un-dogmatic.

PC DOS Pascal has many extensions to Standard Pascal, such as conditional compilation, loop terminators, separate compilation facilities (including a Modula-2 look-alike called a "unit"), an optional pseudo-code listing, and a variety of additional data types. Also provided are built-in functions that access PC DOS system services.

Differences between the two versions are noted where they're important. Also noted, most of the time, are deviations from standard Pascal. Because the language's extensions are in large part the subject of the book, many deviations go unremarked. The author spends some time educating the reader in the uses of DOS, as well as Pascal, though without dissecting the operating system on an assembler level. Little time is spent on running the compiler (presumably covered in the manual) and the linker (also covered elsewhere).

The presentation of the book is good. It's slim, can be made to lie flat. Pascal keywords occurring in the text are printed in typewriter font so they can be picked out quickly when scanning. There is a summary at the top of each chapter, and the author encourages the reader to skip chapters in which the material seems too familiar.

Since all of the information exists in other forms, the strength of the book lies in how well its author anticipates the interests of his readers. I think he's done a remarkable job. The emphasis is on pragmatic information. The "Assembler Interface" chapter, for example, does not attempt to teach the reader PC assembly language, but it does provide the simplest possible assembler routine callable from Pascal.

The book's weakness is that it is (necessarily) sketchy in some areas that may be of interest to many programmers, and the level of understanding at which the book aims doesn't seem consistent throughout. There is sometimes a remarkable lack of precision in order to accommodate the novice reader, as in this description of `char` type variables:

"Char variables ... take values which are letters, punctuation marks, digits, spaces, and in general the sort of thing that one writes down."

The screen handling chapter is a good example of the author's approach Mr. Chess sets forth some reasons for devoting attention to the manner in wh a program displays its output, then lau ches into describing four methods for doing so. These are: standard Pascal readln and writeln statements, ANSI.S' calls, BIOS calls, and writing directly tc screen memory. Each section is illustrat with specific examples. Finally, he offe "Some Speed Considerations," in whic he mentions the tradeoffs involved in selecting one method over another. The quantity of technical material presente surprising, especially considering that t chapter begins with the sentence "The display screen is the PC's voice."

The goal of the book, though the author didn't state it this way, seems to to get the Pascal applications program up and running on his IBM PC as quic as possible. The reader, in theory, has access to all of this material in the form several separate manuals for the compi the linker, and the operating system. M Chess relates these various parts of the development environment to each othe ways that most programmers will appreciate.

The book's strength lies in how well its author anticipates the readers' interests

David M. Barnes is a technical writer w came to Oregon Software via Compu-Serve Incorporated, where he develope interests in microcomputers and softw tools.

Calling VMS system service and run-time library routines via Pascal—2

by Thomas E. Hanrahan

VMS system service and run-time library routines perform a wide variety of functions. They control resources available to processes, provide access to process information, permit communication between processes, manipulate character strings, and allow greater flexibility of input and output operations. Many of these procedures are useful programming tools and are available to Pascal programs.

Both the Pascal—2 compiler and support library running under VMS are written primarily in Pascal and rely on calls to VMS routines. One example is the support library procedure p2_get_foreign, used to read input directly from a command line. P2_get_foreign calls the VMS run-time library routine lib$get_foreign to return the contents of the foreign command line that invokes the compiler.

When you call a VMS system service or run-time library routine from a Pascal program, you must supply whatever arguments and calling sequence the utility requires. In all cases, VMS routines follow the standard VAX-11 conventions for calling procedures, so they must be declared as nonpascal when called from a Pascal program. To determine the arguments that a particular VMS routine requires, you must refer to the VAX/VMS manual in which the routine is documented, usually either the *VAX/VMS System Services Reference Manual* or *VAX-11 Run-Time Library Reference Manual*.

In the case of lib$get_foreign, the VMS run-time library routine follows the format:

where:

- *get-str* is a string to receive the text of the command line.
- *user-prompt* is a string to be used as a
- *rtn-len* is an integer variable passed by reference to return the length of the command line string.
- *force-prompt* is an integer variable set equal to 1 or 0 according to whether you want a prompt issued in all cases or only when command-line text is unavailable.

String arguments are generally passed to VMS routines by reference to VMS "string descriptors." Such string descriptors consist of four components:

- A word containing the length of the string.
- A byte specifying the data type of the argument.
- A byte specifying the class of the argument.
- Four bytes (described by DEC as a "longword") containing the address of the string.

Data-type and class values can be selected from tables provided in the *VAX-11 Run-Time Library User's Guide.*

A VMS string descriptor is defined in Pascal as a packed record whose fields correspond to the descriptor components just described. You begin setting up a string descriptor by defining the string buffer that the descriptor is to reference as some packed array of char. Next, you define two integer-subrange types that fit exactly into a byte (0..255) and a word (0..65535). And finally you define the descriptor itself, as shown in the type declaration of the example on the following page. Notice that pointers generated by the Pascal—2 compiler, such as ^cmdbuffer in the example, are always four bytes in length and may serve as the fourth component of a string descriptor.

VMS system service and run-time library routines may be defined as either procedures or functions, depending on whether or not you want to test for successful completion of the called routine. When declared as a function, the fatal completion.

In p2_get_foreign, the VMS routine lib$get_foreign is defined as a function with four parameters corresponding to the four required arguments for the routine.

The body of the procedure p2_get_foreign (not shown) is devoted to assigning appropriate values to the arguments that must be passed to lib$get_foreign. For instance, the descriptor txt_descr, references the string that returns the command line text, as follows:

```
with txt_descr do
  begin
  len := h2-l2 + 1;
  class = 1;{ fixed-length string }
  dtype: = 0;
  txt_ref := ref (txt_buf);
  end;
```

The descriptor prompt_descr is similarly assigned values. The string prompt_buf, which prompt_descr describes, is assigned the character string passed to p2_get_foreign from the calling procedure or main program. And the variable force is set to 0 because p2_get_foreign returns a prompt only when command-line text is unavailable.

Once assignments have been made to the various descriptors and variables, lib$get_foreign is called by the statement shown in the example. The routine stores the text taken from the command line in the string txt_buf. The contents of txt_buf can then be copied to the string txt by p2_get_foreign and subsequently passed back to the main program or calling procedure. There, the text can be parsed and processed as needed.

The usefulness of the procedure p2_get_foreign is not limited to passing command line text to the Pascal—2 compiler. As part of the Pascal support library, p2_get_foreign can be called from any Pascal program running under VMS as a foreign command. (See the section entitled "Reading Command Lines" in the recently released Update

The ability of Pascal—2 to create an interface such as this one between p2_get_foreign and lib$get_foreign is an important feature of the compiler, allowing you access to the full capabilities of the VAX/VMS programming environment.

```
const
  cmdlinelength = 512;

type
  cmdindex = 1..cmdlinelength;
  cmdbuffer = packed array [cmdindex] of char;
  byte = 0..255;
  word = 0..65535;
  descriptor = ------------------------------------------ VMS string descriptor
      packed record
        len: word;
        dtype: byte;
        class: byte;
        txt_ref: ^cmdbuffer; ------------------------- pointer filling a longword
      end;

function lib$get_foreign(var line1: descriptor;
                         prompt: descriptor;
                         var length1: word;
                         var forceprompt: integer): integer;
  nonpascal; ------------------------------------------ creates correct calling sequence

procedure P2_get_foreign(prompt: packed array [l1..h1: integer] of char;
                         var txt: packed array [l2..h2: integer] of char;
                         var length: integer);
  external;


procedure P2_get_foreign;

  var
    ret_status: integer;
    txt_descr: descriptor;
    txt_buf: cmdbuffer;
    txt_length: word;
    prompt_descr: descriptor;
    prompt_buf: cmdbuffer;
    i, k: integer;
    force: integer;

    begin
      :
      ret_status := lib$get_foreign(txt_descr, prompt_descr, txt_length, force);
      :
    end;
```

This code segment from the Pascal support library routine p2_get_foreign *shows in Pascal terms the definition of a VMS string descriptor and a call to the VMS run-time library routine* lib$get_foreign.

Errors, additions to manuals

Roughly every six months, we distribute update packages along with the release of new versions of our software. These update packages are made up of "change pages" documenting changes and additions to be included in each manual. In some cases, the changes describe new features or enhancements implemented in the software. Other times, the emendations are based on information or requests that we have received from readers through Documentation Evaluation Reports (the DER forms at the back of each manual), Trouble Reports, and support calls. (See the list of update packages issued to date.)

The cycle of update packages is staggered and generally depends on a product's original release date. During the interim between updates, we list changes and additions in this newsletter and the Release Notes. Note: whenever a change is made in response to a Trouble Report, we've placed the Trouble Report number in square brackets at the end of the entry.

Oregon Software Update Packages

Update packages are issued at roughly every other release of the software, with changes to intermediate releases being documented by Release Notes and this newsletter. Note: not all products are initially released as version A; Pascal-2 for VMS, for example, began as version 2.1C.

User Manual	Publication Date
Pascal—2 V2.1 for RSX	July 1983
Update Package No.1 V2.1B	October 1983
Update Package No.2 V2.1D	February 1985
Pascal—2 V2.1 for RSTS	August 1983
Update Package No.1 V2.1B	October 1983
Update Package No.2 V2.1D	February 1985
Pascal—2 V2.1 for RT	August 1983
Update Package No.1 V2.1B	October 1983
Update Package No.2 V2.1D	February 1985
Pascal—2 V2.1 for VMS	October 1984
Update Package No.1 V2.1D	May 1985
Pascal—2 V2.1 for UNIX (68000)	August 1983
Update Package No. 1 V2.1D	April 1984

All Pascal—1 manuals

Our April release of the 1.3D Pascal—1 software coincided with our distribution of the 3rd edition of the Pascal—1 User Manual. Like the 1.3D software, the Pascal—1 User Manual has become an unsupported product. Whenever possible, however, we will publish corrections or additions to the manual in the *Pascal Newsletter*, and we encourage Pascal—1 users to send us tips or descriptions of problems that we may pass along to other Newsletter readers.

All Pascal—2 manuals

Page numbers are not provided in this section because they vary from book to book. Instead, the location for changes is described by section name and paragraph or line number where appropriate.

Constants not included in "Dead Code" elimination

Under the description of "Dead Code Elimination" in the section "Compiler Optimization," add the following sentence to the second paragraph:

> The compiler does, however, generate constants in the constants' psect for string literals that appear in write and writeln statements within a dead code region.

[2425]

PASMAT won't format %include files

In the section "PASMAT: A Pascal—2 Formatter," add to the list "Limitations and Errors" the following item:

- PASMAT does not process %include files. Consequently, if the A directive is specified, an identifier is determined by that of its initial occurrence in the file being formatted and not by its form in the %include file.

[2201]

All Pascal—2 manuals for PDP-11 and Professional 300 Operating Systems

Embedded switch limitation

In the "Embedded Switches" section of the "Programmer's Guide," add the following sentence to the beginning of the fifth paragraph:

> You may use up to 25 embedded switches in a compilation unit.

[2760]

Extended-range arithmetic

Replace the first paragraph of the "Extended-Range Arithmetic" section of the "Language Specification," with the following text:

> The normal range of integer variables in Pascal—2 is -32767..32767, but you may also declare variable types in the extended range of 0..65535. A variable with an upper limit greater than 32767 is called an extended-range or "unsigned" variable. Any integer value may be assigned to an unsigned variable and is converted as a bit pattern. If the value being assigned is negative, the error is not trapped at run time, since there is no way for the compiler to tell the difference between a negative "signed" value and an extended-range "unsigned" value. The same sort of implicit transformation is true when an extended-range value is assigned to an integer variable.
>
> The arithmetic operations of addition, subtraction, and multiplication are always signed operations. They treat both signed and unsigned variables as though they were signed. The results for these operations on signed variables are always correct; results produced for unsigned variables are correct except in overflow conditions involving multiplication. Division and modulo are unsigned operations for dividends in the extended range, but they do not treat divisors greater than 32767 as unsigned values. Comparison operations are signed or unsigned according to variable type.

Pascal—2 for RSX

Overlay structures, pages 2-18 and 2-19

Replace the last sentence in the description of the DEBUG2 structure with:

> For programs using single-precision, you have to specify SING, a special variant of the SINGLE co-tree, designed for use with DEBUG2. SING accounts for the fact that the Debugger is itself a Pascal program compiled with double precision.

In the third overlay example on page 2-19, change the SINGLE co-tree to read SING. [2803]

Stack overflows, page 2-31

Add the following text immediately before the section entitled "The 'Space' Function."

Caution

> Very large stack overflows, those that extend beyond the heap and into the program code sections of memory, have the potential for causing serious problems. In some cases it is possible for the error to prevent the appropriate error message from being printed or the program from properly terminating. In some rare instances, the condition can even lead to the disruption of the computer's operating system. It is the programmer's responsibility to avoid such excessively large overflows by controlling the size of local variables and value parameters passed to procedures.

Error walkback with I & D space programs, page 2-42

At the the bottom of the page, insert the following paragraph:

> Error walkback cannot be used with Instruction & Data (I & D) space programs available on RSX-11M/PLUS. Normally (in non-I & D space programs), the walkback routine examines instructions to find special markers and pointers to data areas, which contain procedure names and statement locations. But, in I & D space programs, instructions and data are kept separate, and this, coupled with the fact that data sections can be overlaid, makes it difficult for the walkback routine to function properly. In some I & D space programs, the walkback routine reports addresses instead of procedure names, but in other instances, the walkback procedure can abort with an odd address error or memory management trap. You should generally compile all I & D space programs with the /nowalkback switch to avoid such problems.

[2516]

Program example, page 2-59

In the program example, change t statement:

```
Efn := 1; {use event flag}
```

to read:

```
Efn := (16*256) or 1
```

The new line checks for all 8 words ir the Status Block. [2800]

Debugging with I & D space programs, page 4-1

At the the bottom of the page, in sert the following text:

Caution

> The Debugger cannot be used on I & D space programs. The separation of instructions and data generated by such programs makes difficult for the Debugger to prope trace procedures and statements. (S "Run-Time Error Reporting" for more details.)

Pascal—2 for RT

Command line error, page 1-3

The command line for running th program CUSTOM is wrong. It shoul be changed to:

```
.R CUSTOM
```

Foreground Operations, page 2-49

The last sentence on the page should be changed to read:

> The period after 1024 signifies a decimal value.

Pascal—2 for UNIX (68000)

A number of pages in the manua contain minor errors that can be best changed by penciling in the changes a described:

Multiple embedded options, page 2-5

Below the sample embedded direc tive lines:

```
{$noindex,norange}
{$noindex}{$norange}
```

Add the sentence: "A space between options in the first command line, i.e. {$noindex, norange}, is not allowed and the compiler treats everything after the space as an ordinary comment. "

Selective debugging, pages 2-5 and 2-6

In several places on these pages, the manual states incorrectly that the debugging option can be turned "on" or "off" for sections within a module. The correct explanation is that the embedded options $debug and $nodebug actually have no effect on the current version of the Debugger for UNIX/68000. Debugging can occur only for an entire compilation unit and may be invoked only by specifying the -debug option on the compilation command line.

Removal of parameters from the stack, page 2-15

The last sentence in the fourth paragraph should read, "It is the responsibility of the calling procedure to remove parameters from the stack."

Storage allocation for packed record, page 2-17

The last sentence in the paragraph should read, "beginning at bit 8 (most significant bit)."

'Read' and 'Readln' procedures, page 3-15

Replace the existing heading with 'Read' and 'Readln' Procedures and add the following text after the section's first paragraph:

> For variables of type subrange, read and readln statements accept values for the variable outside the limits defined for the subrange and the error is not recognized until such values are actually used. Error checking at the time of input, for values outside the limits of a subrange, is the programmer's responsibility.

Program listing, page 3-21

The type declaration of Word is incorrect in the program listing. The correct limits should be defined as 0..4294967295 and the program shown in the example.

Process termination values and statements, page 4-2

Add the following paragraph just before the final paragraph on the page:

> A process termination (exit) value and statement is given by the Debugger each time a program terminates, whether termination is normal or the result of some error. Exit values are either 1 or 0. An exit value of 1 always indicates that termination is the result of a run-time error and is preceded by the appropriate run-time error message. An exit value of 0 indicates either normal termination or termination because of some error other than run-time, such as a bus error. Except for normal terminations, error messages are also given for terminations that yield exit values of 0.

Passing parameters in the Debugger via Go, page 4-10

Replace the existing description for the G: Go command with the following:

> The G (Go) command begins executing the program at MAIN.1; this command may be used at any point to restart the program. See the example after the C command.
>
> If you debug a program that requires files to be passed as arguments, such files can be passed as parameters (similar to those in a write statement) with the G(*'file'*) command. For example, suppose that the program test.pas requires that data.dat be passed at some point during program execution. Compile test.pas and invoke the Debugger with:
>
> pc -debug -output test test.pas
> pdb test
> }
>
> Use the G command to pass data.dat by entering:
>
> } g('data.dat')

Note that the argument is enclosed in single quotes (' '). Multiple arguments quotes and separated by commas, as:

} g('arg1','arg2','arg3',...)

'For' statement index entry is incorrect, page Index-2

The correct page reference for for statements is 3-28.

Pascal—2 for VERSAdos

The manual, soon to be released as a new edition, is being thoroughly reformatted to conform to the style of our other Pascal—2 manuals. The text contains minor corrections and additions throughout. Appendices listing compilation and run-time errors will be brought up to date.

Pascal—2 Cross-Development System

An update package is being prepared that reflects the switch from the OASYS cross-linker/assembler to the Oregon Linker/Assembler. The update package will also include an index for the first time.

Continued from Page 1

and several cross-compiler configurations. We will provide more details later.

Other developments

Maintenance work is continuing on our other products. The latest update for our VAX/VMS native compiler, version 2.1D, was released in early June.

Technical problems in the first shipments of version 2.1D of Pascal—2 for the PDP-11 caused us to pull back that release. The problem has been corrected, and customers who received the faulty shipments have been updated. All shipments of Pascal—2 for RSTS, RSX, and RT-11 are now Pascal V2.1D. Work on 2.1E will begin this summer.

The Log: errors, work-arounds, changes

This log describes significant changes in Oregon Software products. As a service to our users who have reported problems to us in the past, we have provided Trouble Report numbers where possible. At the end of this section, we've also listed reported bugs that have been verified by us as being problems with the software. Where possible, work-arounds for know problems are provided after the description of the error.

All Pascal—1 compilers

The following is a list of problems corrected by the 1.3D release.

Number	Description
TR 508 TR 1213	The assignment of a value to a function identifier went undetected. Now, the compiler generates the error message `Illegal assignment`.
TR 845 TR 1357	The compiler failed to detect an illegal expression containing two consecutive operators. Now, it issues the error message `Bad expression`.
TR 897a	Pascal—1 incorrectly evaluated boolean operations on integers with magnitudes approaching `maxint`. Such expressions are now correctly evaluated.
TR 899 TR 1262 TR 1993	External procedures declared twice in the same program caused the compiler to crash. Now, the error message `Bad procedure name` is issued.
TR 1149	Expressions involving combinations of `pred`, `ord`, and `succ` functions occasionally returned incorrect results. Such expressions are now correctly evaluated.
Unassigned	The `get` procedure no longer fails when accessing a file of type `text`.

Pascal—2 for VMS

The following is a list of bugs fixed in version 2.1D.

Number	Description
TR 2513	The Pascal—2 command line prompt displayed two different strings. A single carriage return generated the string `PAS>` ; a second carriage return resulted in `MP2>`. The string `PAS>` is now displayed in all cases.
TR 2826	The installation command file, `INSTALL.COM`, failed to create the Pascal—2 help library when an existing library structure was not already in place. Documentation has been added to the Installation Guide describing steps to take in such instances.
Unassigned	The I/O control switch `/temp` failed to deallocate space used by temporary files, even though it deleted the corresponding file directory entries. The `/temp` switch now correctly deletes temporary files.

Set problems corrected

TR 2501 TR 2511	Defining a set as a structured constant caused the compiler to abort with an access violation. The compiler now generates the correct syntax-error messages for such conditions.

TR 2527
TR 2548

Certain set instructions failed to produce the correct code and in some cases resulted in an access violation.

TR 2549

The compiler failed to generate an error message when a variable was declared to be a set of an enumerated type consisting of more than 256 elements. Such set declarations now produce the compilation error message Set types must have base in the range 0..255.

TR 2759

Set expressions with non-scalar elements, such as strings, caused the compiler to crash.

Conformant array bugs fixed

TR 2465

Certain programs hung when a call was made from within a for loop to a procedure with a conformant array parameter. Such programs now execute correctly.

TR 2526

Undefined variables passed to conformant array parameters of a procedure or function were not trapped by the compiler and resulted in access violations. Such variables are now properly labeled by the compiler as undefined identifiers.

TR 2574
TR 2602

Passing a structured constant by reference to a conformant array parameter resulted in the error message Too many nodes in procedure. The compiler now generates the correct compilation error Variable name expected.

Readin', writin', and . . .

TR 2555

Integers written to a file of real were not converted to real number format. This led to a run-time error when an attempt was later made to read such files. The compiler now generates code that converts integers to reals when they are written to a file of real.

TR 2713b

Attempts to read the real number 0.0 caused programs to loop indefinitely. The compiler now generates code that correctly reads the value 0.0.

. . . 'Rithmetic problems fixed

TR 2651
TR 2720

Incorrect values were returned for mod operations performed on negative numbers. Mod now generates results according to the Pascal standard.

TR 2815

The absolute value function, abs, incorrectly used a source register for a destination register. As a result, a variable with a negative value was changed to positive when it was used as an argument of abs. The compiler now processes absolute values correctly.

Unassigned

The mod function returned incorrect results for unsigned integers when both the dividend and divisor were greater than 2147483647.

'Reset' and 'rewrite' bugs fixed

TR 2576

Certain attempts to reset the standard file input caused a run-time error. The statement reset(input) now performs correctly.

TR 2616

Text was stored in a file improperly when an attempt was made to rewrite the file after it had already been opened for writing. The compiler now generates code that correctly resets a pointer to the file's buffer.

Debugger problems corrected

TR 2684

When a command line included both file extensions for output files and the /debug switch, as shown in the example:

```
$ pas2 main.obj,main.lis = main.pas/debug
```

the VAX Linker detected illegal record sequences in the resulting object file. The

Unassigned	Variables stored in registers by the compiler returned incorrect values when watched or written by the Debugger. The Debugger now handles variables stored in registers correctly.
Unassigned	Variables in the last line of a program were not previously accessible to the Debugger. Such variables are now available.
Unassigned	The "watch" Debugger command W could not be set before the first line of a program was executed. With this release, watched variables can be specified before you begin execution of a program.

Pascal—2 for UNIX (68000)

Version 2.1G corrects the following problems with the compiler.

Number	Description
TR 2291	The system call getpid, a C function declared as nonpascal in a Pascal program and used to return a program's process ID number, caused the run-time error Illegal instruction (core dumped) whenever it was called from within a write or writeln statement. The system call now functions properly within write and writeln statements.
TR 2666	Mod operations computed incorrect results when performed on variables passed as parameters to a procedure or function. Such operations now generate correct values.
TR 2667	The compiler interpreted file names to be all lower case letters when searching for %include files. As a result, lookup failed on %include files whose names contained upper cases letters, such as Test.pas. The compiler is now case sensitive in searches for %include file names and recognizes file names with upper case letters.

Problems with arrays corrected

TR 1917

The compiler crashed whenever a variable of type subrange was used as the bound in an array declaration, as with the variable z in the example:

```
type
  a = 1..7;
  x = 1..7;
  y = 1..7;

var
  z: a;
  ex: array [x,y,z] of integer;
```

The compiler now generates the correct compilation error message Bad constant in such cases.

TR 1961
TR 2172
TR 2531

The run-time error message Integer overflow resulted from illegal type declarations of the form type T = array [1..const-1], where an expression rather than a constant was used to define the limit of an array. The error is now trapped during compilation and identified with the proper syntax error message: ',' expected.

TR 2660

Certain Subscript out of bounds errors were incorrectly identified as variable subrange exceeded errors. The problem occurred when a variable of type subrange was used as the index of a packed array and a value was assigned to the variable that exceeded the bounds of the index but not the limits of the subrange. The correct Subscript out of bounds error message is now given in such cases.

TR 2679

Changing the value of a single element in a packed array that was a field of a packed record caused other elements of the array to be similarly modified. Such operations on packed arrays within a packed record are now performed properly.

Known Bugs

We haven't completed our work on all the Trouble Reports that we have received; the following is a description of those problems we've verified. Except for the problems noted with the now frozen Pascal—1 compiler, the bugs listed here are all scheduled for correction in subsequent releases.

Pascal—1

The following is a list of bugs outstanding in 1.3D.

Number	Description
TR 465	When calling FORTRAN library functions using the `fortran` procedure option and the `/X` compilation switch (double precision reals), the function return value is lost.
TR 846	Double-precision function calls occasionally leave the stack in an inconsistent state.
TR 968	IMP-processed code generates a `Branch out of bounds` error during assembly.
TR 1069	For the RSTS compiler, the `reset` procedure uses an assigned account number as the default directory.
TR 1348	Assignment from a function passed as a parameter to a procedure generates an `Incompatible type` error.
TR 1552	A recursive procedure with a procedure parameter does not compile.
TR 1811	The compiler fails to detect the second occurrence of an illegal operand.
TR 2174	The compiler occasionally fails to diagnose a procedure call containing an incorrect number of parameters.
TR 2267	Some file handling operations (`Close`, `Reset`, `Rewrite`) may generate spurious code unless followed by a semicolon at their call sites.
TR 2330	In a program with nested procedures, a `goto` from within a `for` loop in the innermost procedure occasionally fails to find its destination in the outer procedure.
TR 2335	When writing to a file from a subscripted variable, the subscript calculation is sometimes incorrect.
TR 2388	When a variable of type `text` is assigned to another variable of type `text` the compiler occasionally fails with an odd address trap instead of issuing an error message.

Pascal—2 for VMS

The following are known problems in version 2.1D.

Number	Description
TR 2514b	The compiler fails to return a file name when it reports a run-time I/O error.
TR 2550	In certain cases, run-time errors that should generate the message `Variable subrange exceeded` are flagged as `Array index out of bounds`.
TR 2610	When specified on the compilation command line, an illegal file name with two consecutive "dots" (i.e., TEST..PAS), causes the compiler to generate the run-time error message `Can't open file`.

TR 2642	In certain cases the compiler fails to recognize an Array index out of bounds condition when compiled with checking enabled. Compiling with the /nocheck switch causes the errors to be correctly identified.
TR 2713a	The function trunc does not check for extended-range values and converts all unsigned real numbers to signed integers.
TR 2744 TR 2837	In certain instances the compiler generates incorrect code for empty case statements with constant case selectors.
TR 2768	The Dynamic String Package procedure insert( ) fails to truncate the target string properly when *max-len* characters is exceeded. Such overflows result in the run-time error Array index out of bounds.
TR 2776	The run-time error message File not open is produced when you use a single readln statement to input a packed array of characters and an integer from a text file, as in: readln(text, str, i); where text is a file of text, str is a packed array of characters, and i is an integer.
TR 2829	Write statements generate spurious carriage returns when escape sequences or VMS run-time library calls are used to format screen output.
TR 2862	Readln(i), where i is of type integer accepts all types of input. As a work-around, use read(i) followed by readln to input integers.
TR 2863	Lack of an argument for the support library routine iostatus generates the error message Too many nodes in the procedure rather than the correct error message pair '(' expected and ')' expected.
TR 2870	The compiler incorrectly parses packed set elements of packed records when the set contains more than 32 members.
Unassigned	The Debugger prompt } is followed by a spurious carriage return, causing your input to be written one line below the level of the prompt. Consequently, a typical Debugger command appears on your screen as: } B(MAIN,5)

Pascal—2 for UNIX(68000)

The following are known problems in version 2.1G.

Number	Description
TR 1961 TR 2241	Incorrect code is generated with certain element assignments of packed arrays of packed records when the record fields involved are sets. Pascal—2's optimizations cause the resulting code to be generated with byte rather than word instructions and lead to incorrect results.
TR 2536	Certain programs that attempt to write elements of arrays that are themselves packed arrays of char return the run-time error message Illegal instruction (core dumped).
TR 2586	A mismatch between a type definition and a structured constant definition using that type causes the compiler to crash and return the error message Illegal instruction (core dumped).

TR 2589

Invoking both of the compiler options -debug and $nolist during a single compilation causes the compiler to generate an incomplete listing file and the Debugger crashes.

TR 2730

Specifying a pointer to a file variable in the argument of a rewrite( ) statement causes a core dump upon termination of the program. The program compiles and runs correctly except for the termination error, and the rewrite( ) statement is properly executed. The core dumped error does not occur when a file variable is passed directly to the rewrite( ) statement. This error was first reported in Version 2.1E.

Problems in the Debugger

TR 2445

The Debugger does not have access to function values because the symbol table file currently only recognizes variable names.

TR 2658a

The Debugger does not write correct values of constants.

TR 2658b

The Debugger fails to return an error message when you to attempt to assign a value to a constant. The assignment, however, is never made, and the value of the constant remains unchanged.

TR 2731

When a run-time error is encountered in a program being debugged, the Debugger is not able to recover the stack frame from which the error occurred and, as a result, it loses access to the program's variables. This condition will be corrected by the introduction of a post-mortem analyzer in a future release. As a temporary work-around, when you encounter a run-time error during debugging, set a breakpoint at the statement in which the error occurs, then restart the program and run it to that breakpoint. Once you reach the statement, you can accurately probe variables immediately before the point at which the run-time error appears.

TR 2732

In modules compiled with the $own option, Debugger commands taking an argument true, false, or nil are flagged as Unknown identifier errors when set within a stack-frame context.

As a work-around, return to the main program, set the command you wish to use, and proceed back to the breakpoint you want to examine. You can accomplish this using the command line:

```
} e(1);H(true)
```

Oregon Linker/Assembler

The Oregon Linker/Assembler has two outstanding bugs at this time.

Number	Description
Unassigned	The linker map fails to display sections containing no code. Sections that only define storage declarations do not appear on the map.
Unassigned	Extraneous spaces, special symbols such as $, and version numbers on the command line are flagged as illegal.

Newsletter Index

This index catalogs topics that have appeared in the *Pascal Newsletter.* Entries are listed by newsletter number (in bold) and page number. Readers' comments on the format as well as the contents of this index are welcome.

A, B

active page registers (APRs),
- controlled address ranges, **9**:4
- savings with virtual overlays, **8**:1

asynchronous system traps (ASTs),
- implementation using DoCmd, **4**:15
- Pascal—1 example, **8**:20

book reviews,
- *Introduction to Pascal,* **3**:3
- *Programming in PC DOS Pascal,* **10**:9

C

checkpointing, **1**:7, **3**:2
cluster libraries, **8**:1
common blocks, **7**:21
Concurrent Programming Package, **7**:8

D

default devices incorrect, **2**:4
detached tasks, **8**:8
device control registers, **8**:7
device drivers, **7**:9
device monitors, **7**:9
disk space problems, under RT/TSX, **3**:16

E

embedded systems, **7**:8
EMT predefined procedure, **2**:5, **3**:6
EXTK$ directive, **3**:2

F

FCSRES, **8**:1
file,
- efficient packing of data, **7**:4
- looking on wrong device under RT/TSX, **2**:3
- out of memory under TSX, **1**:7
- overflow error, **4**:21
- random access, **6**:8
- speeding up opening process, **5**:5
- unformatted FORTRAN structure, **7**:4

file variable under RT/TSX, **4**:18
FMS-11, **8**:20
FORINI, initialization routine, **4**:22
for loop restriction, **3**:7
FORTRAN,
- called from Pascal—2, **4**:21, **4**:22
- calling Pascal—1, **1**:5
- FORINI routine, **4**:22
- I/O restrictions, **4**:21

FORTRAN carriage control, **6**:5
forward directive, **3**:6

G, H

getpos, *see* random access
global symbols, access from Pascal programs, **7**:7
$$heap,
- cross-reference, **2**:3
- extension of, **3**:2

I

I/O channel numbers, retrieval of, **4**:18
I/O control switches,
- enhancing terminal performance, **8**:11
- implementing single-character I/O under RSX, **6**:4
- reading unformatted FORTRAN files, **7**:5
- using FORTRAN carriage control under RSX, **6**:5

I/O devices, access from Pascal, **8**:7
I/O errors, **6**:6
- error trapping, **6**:7
- printing under RSX, **6**:6
- sayerr procedure, **6**:7

I/O page, **8**:7
integers,
integers, extended range, **6**:11
- overflow/underflow, **3**:6
- undetected overflow, **3**:7
- unsigned, **6**:11

interactive I/O, **6**:10
interrupt service routines, **4**:10
- CPP, **7**:8
- WAITIO, **7**:12
- interrupt vector initialization, **4**:10

K, L

kernel,
primitives, **7**:11
structure, **7**:15
lazy I/O, **6**:10
synchronization of I/O, **6**:10
literal strings, use in structured constants, **5**:8
loophole function, **2**:4, **9**:6

M

mapping virtual to physical addresses, **9**:4
memory management hardware, **9**:4
memory organization,
under Pascal—1, **7**:13
under RSX, **3**:1, **9**:4
under RT-11, **4**:7
menu-driven modeling, **3**:7
monitor error messages, **1**:6

N

networking program, **3**:7
newstart routine, *see* User Service Routine (USR)
nluns, double definitions, **4**:22
nonpascal directive, **4**:22
fortran renamed nonpascal, **2**:3
not enough memory,
under RSX, **3**:1
under RT/TSX, **3**:16

O

OPUS, **5**:2
Communique, **5**:2, **6**:14, **7**:7, **8**:22, **9**:3
library, **7**:7
membership list, **9**:17
Oregon Pascal User's Society, *see* OPUS
origin, **2**:3, **2**:4, **4**:23
CPP, **7**:9
restrictions, **4**:23
use with common blocks, **8**:21
overlays, **8**:1
NODSK attribute, **5**:7
under RSX, **3**:3, **5**:7
under RT/TSX, **5**:4
virtual, **8**:1
overprinting, *see* FORTRAN carriage control

P, Q

partitions, , **9**:4
allocation example, **9**:5
creation of, **9**:4
for resident libraries, **8**:1
PAR utility, **8**:1, **8**:7, **9**:4
Pascal—1, **3**:5
called from FORTRAN, **1**:5
comparison with Pascal—2, **2**:3, **3**:5
compiler crashes, **1**:6
real-time facilities, **7**:11
text formatting, **3**:9
variable length strings, **3**:11
Pascal—2, **3**:1
calling FORTRAN routines, **4**:21, **4**:22
comparison with Pascal—1, **2**:3
summary of new features, **6**:3
Pascal programming tools, **9**:3
MEASURE, **5**:3
PVS Quality Control Package, **9**:3
Standard Pascal Mode Implementation, **9**:3
Syntax Checker, **8**:4
PASLIB, **3**:2
PASRES, **8**:1
PLAS directives, **2**:3, **8**:1
portable database system, **3**:7
preventing task extension, **2**:3
privileged tasks, **8**:8
process-control systems, **8**:17
process-monitor model, **7**:9
PSECTS, **3**:2
Quick Task Builder, **4**:2

R

random access, to text files, **6**:8
random number generator, **4**:3
record locking under RT/TSX, **4**:18
record management systems, **9**:3
Pascal Record Management (PRM-11) **9**:3
resident libraries, **1**:8, **8**:1
RMS-11K, **4**:2, **8**:20
ROM applications, **4**:7
cross-reference, **5**:8

S

sayerr procedure, *see* I/O errors
/seek, **3**:6
semaphores, **7**:12
SET/MAXEXT, **3**:2
setpos, *see* random access
SETSIZ program, **3**:16
shared date, **9**:4
shared tasks, **9**:4
single-character I/O, **6**:4
SJ monitor, restrictions, **4**:21
spawning sub-tasks, **4**:15
SQUEEZE command, **3**:16
stack, overflows, **3**:2

stand alone programs, **4**:7
start,
    failure under RSX, **1**:7
    failure under RT/TSX, **2**:4, **3**:16
    use of $START, **3**:16
static local variables, **2**:8
surveys
    consumer survey, **5**:11
    Modula-2 interest, **9**:16
Syntax Checker, **8**:4
SYSLIB, **3**:2
systems calls,
    from RSTS, **2**:5
    from RSX, **4**:15, **8**:19

T

Task-builder, **3**:2, **9**:6
task extension, **3**:2
task image, reducing size, **5**:7, **8**:1
terminal I/O, **6**:4
transfer address, *see* transfer symbol
transfer symbol, address incorrect, **4**:21
    incorrect for RT/TSX, **2**:4, **3**:16

U, V, X

unsigned integers, *see* integers
User Service Routine (USR),
    reserving memory, **5**:5
variable length strings, *see* Pascal—1
variables,
    detecting initialized variables, **3**:7
    XREF utility, **3**:7
VMR utility, **9**:5
XM monitor, virtual jobs, **5**:4
XREF utility, **3**:7

Pascal
NEWSLETTER
OREGON SOFTWARE

6915 S.W. Macadam Avenue
Portland, Oregon 97219

DISTRIBUTOR CUSTOMERS

Please submit all technical support questions, verbal or written, directly to your authorized Oregon Software Distributor.

When you have a problem, please contact your Distributor by telephone. If the problem cannot be solved over the telephone, we request that you submit a written Trouble Report.

When submitting a written report, please complete one of the enclosed Trouble Report Forms and send it to your Distributor along with a small sample program demonstrating the problem you are experiencing. If you cannot reduce the problem to a one-page sample program. Submit your Trouble Report with the program on a single density floppy disk (preferably RT-11 format) or on a magtape. Your Distributor will then duplicate the problem at his facility.

In most cases your Distributor will be able to answer your questions. If necessary, your Distributor will forward your written report to Oregon Software for a solution. We will do everything possible to respond to your problem within 30 days after the report is received at Oregon Software. Once a solution is reached, it will be forwarded to your Distributor who will then contact you.

Thank you for support and interest in Oregon Software products.

Trouble Report

See instructions on reverse side.

CUSTOMER IDENTIFICATION

Your Name _____ Designated Contact Person (if different) _____

Conditional Update Request: If the Trouble Report pertains to a problem that has been fixed or will be fixed in the next release, this signature constitutes a request for an update which Oregon Software will provide under the terms of the existing software license and/or support agreements.

Signature _____ Date _____/_____/_____

Company _____ License Number _____

Address _____ Date Submitted _____/_____/_____

_____ Telephone ( ) _____ ext. _____

City _____ State _____ Country _____ Zip/Postal Code _____

Processor Model _____ Processor Options (EIS, FIS, FPP, FPA, etc.) _____

Distribution Medium _____ Attachments (Listings, Magtape, etc.) _____

Software and Version _____ Operating System and Version _____

PROBLEM DESCRIPTION

( ) Major Problem ( ) Minor Problem ( ) Suggestion ( ) Inquiry

If the problem could have been avoided by better documentation, explain below.

If the problem is intermittent, outline below the conditions under which the problem seems to occur.

Describe the problem (use additional sheets if necessary):

FOR OREGON SOFTWARE USE

Log Number _____ Title _____

Date Assigned _____/_____/_____ Problem Type _____

Assigned to _____ _____

Date of Response to Customer _____/_____/_____ Summary of Action _____

( ) Circulate _____

Oregon Software
6915 S.W. Macadam Avenue
Portland, Oregon 97201, USA
503/245-2202
TWX: 910-464-4779

INSTRUCTIONS

Check recent issues of our Pascal Newsletter to see whether the trouble has been reported. The Newsletter may describe a quick fix or tell you that the problem has been fixed in the current release. If so, your Designated Contact Person should request an update in writing. If our Newsletter and manuals don't help, complete the front of this Trouble Report and send it to Oregon Software. A few suggestions:

Your Name: We'll call or write you if we need more details about the problem.

Designated Contact Person: The "responsible official" for our software at your site, usually the senior programmer. We'll send the DCP a copy of the correspondence about this problem.

License Number: The serial number of your site printed in the headline of all program listings. Looks like "1-876."

Some problems relate to specific combinations of hardware, software, and processors. We need to know your setup:

Processor Model: The processor line and specific model number (*not* the serial number!). Examples: *PDP-11/44, 256 KB* • *LSI-11/23, 56 KB* • *VAX-11/750, 768 KB.*

Processor Options: Any installed processor options that extend the instruction set. EIS = Extended Instruction Set. FIS = Floating Instruction Set. FPP = Floating-Point Processor. FPA = Floating-Point Accelerator.

Operating System and Version: The software operating environment. Please, *please*, include the specific version! Examples: *RT-11 V4 XM* • *RSX-11M-Plus V3.2* • *RSTS/E V7A* • *VMS V2.1*

Problem Description: Reduce the problem to the simplest situation in which it occurs. One page is the largest program we can reasonably type by hand. If a larger amount of code is needed to demonstrate the error, send us a magtape or single density floppy. Unfortunately, we cannot return these. Note any intuitions you have about possible causes of the trouble. Remember: WE MUST BE ABLE TO REPRODUCE THE TROUBLE HERE.

OUR RESPONSE

It's already fixed or *Fixed in next release.*

You will automatically receive a copy of a release in which the problem is fixed, if your Designated Contact Person has signed the Conditional Update Request on the front of this form.

Maybe a bug, but ...

Documentation problems, customer errors, obscure "features" or inherent limitations in our implementation can each cause trouble. We will help you work around the problem and/or supplement our documentation in these cases. Please understand that our support is limited to correcting problems with our software and does not include applications programming.

In any case—

We will send you a written response within a month.

Trouble Report

See instructions on reverse side.

CUSTOMER IDENTIFICATION

Your Name _____ Designated Contact Person (if different) _____

Conditional Update Request: If the Trouble Report pertains to a problem that has been fixed or will be fixed in the next release, this signature constitutes a request for an update which Oregon Software will provide under the terms of the existing software license and/or support agreements.

Signature _____ Date _____/_____/_____

Company _____ License Number _____

Address _____ Date Submitted _____/_____/_____

_____ Telephone ( ) _____ ext. _____

City _____ State _____ Country _____ Zip/Postal Code _____

Processor Model _____ Processor Options (EIS, FIS, FPP, FPA, etc.) _____

Distribution Medium _____ Attachments (Listings, Magtape, etc.) _____

Software and Version _____ Operating System and Version _____

PROBLEM DESCRIPTION

( ) Major Problem ( ) Minor Problem ( ) Suggestion ( ) Inquiry

If the problem could have been avoided by better documentation, explain below.

If the problem is intermittent, outline below the conditions under which the problem seems to occur.

Describe the problem (use additional sheets if necessary):

FOR OREGON SOFTWARE USE

Log Number _____ Title _____

Date Assigned _____/_____/_____ Problem Type _____

Assigned to _____ _____

Date of Response to Customer _____/_____/_____ Summary of Action _____

( ) Circulate _____

Oregon Software
6915 S.W. Macadam Avenue
Portland, Oregon 97201, USA
503/245-2202
TWX: 910-464-4779

OS-116

INSTRUCTIONS

Check recent issues of our Pascal Newsletter to see whether the trouble has been reported. The Newsletter may describe a quick fix or tell you that the problem has been fixed in the current release. If so, your Designated Contact Person should request an update in writing. If our Newsletter and manuals don't help, complete the front of this Trouble Report and send it to Oregon Software. A few suggestions:

Your Name: We'll call or write you if we need more details about the problem.

Designated Contact Person: The "responsible official" for our software at your site, usually the senior programmer. We'll send the DCP a copy of the correspondence about this problem.

License Number: The serial number of your site printed in the headline of all program listings. Looks like "1-876."

Some problems relate to specific combinations of hardware, software, and processors. We need to know your setup:

Processor Model: The processor line and specific model number (*not* the serial number!). Examples: *PDP-11/44, 256 KB • LSI-11/23, 56 KB • VAX-11/750, 768 KB.*

Processor Options: Any installed processor options that extend the instruction set. EIS = Extended Instruction Set. FIS = Floating Instruction Set. FPP = Floating-Point Processor. FPA = Floating-Point Accelerator.

Operating System and Version: The software operating environment. Please, *please*, include the specific version! Examples: *RT-11 V4 XM • RSX-11M-Plus V3.2 • RSTS/E V7A • VMS V2.1*

Problem Description: Reduce the problem to the simplest situation in which it occurs. One page is the largest program we can reasonably type by hand. If a larger amount of code is needed to demonstrate the error, send us a magtape or single density floppy. Unfortunately, we cannot return these. Note any intuitions you have about possible causes of the trouble. Remember: WE MUST BE ABLE TO REPRODUCE THE TROUBLE HERE.

OUR RESPONSE

It's already fixed or *Fixed in next release.*

You will automatically receive a copy of a release in which the problem is fixed, if your Designated Contact Person has signed the Conditional Update Request on the front of this form.

Maybe a bug, but ...

Documentation problems, customer errors, obscure "features" or inherent limitations in our implementation can each cause trouble. We will help you work around the problem and/or supplement our documentation in these cases. Please understand that our support is limited to correcting problems with our software and does not include applications programming.

In any case—

We will send you a written response within a month.